A Simpler Variant of Universally Composable Security for Standard Multiparty Computation

Ran Canetti[*] Asaf Cohen[†] Yehuda Lindell[‡]

November 20, 2014

Abstract

In this paper, we present a simpler and more restricted variant of the universally composable security (UC) framework that is suitable for "standard" two-party and multiparty computation tasks. Many of the complications of the UC framework exist in order to enable more general tasks than classic secure computation. This generality may be a barrier to entry for those who are used to the stand-alone model of secure computation and wish to work with universally composable security but are overwhelmed by the differences. The variant presented here (called simplified universally composable security, or just SUC) is closer to the definition of security for multiparty computation in the stand-alone setting. The main difference is that a protocol in the SUC framework runs with a *fixed set of parties* who know each other's identities ahead of time, and machines *cannot be added dynamically* to the execution. As a result, the definitions of polynomial time and protocol composition are much simpler. In addition, the SUC framework has authenticated channels built in, as is standard in previous definitions of security, and all communication is done via the adversary in order to enable arbitrary scheduling of messages. Due to these differences, not all cryptographic tasks can be expressed in the SUC framework. Nevertheless, standard secure computation tasks (like secure function evaluation) can be expressed. Importantly, we show a natural security-preserving transformation from protocols in the SUC model to protocols in the full-fledged UC model. Consequently, the UC composition theorem holds in the SUC model, and any protocol that is proven secure under SUC can be transformed to a protocol that is secure in the UC model.

Keywords: universal composability, secure multiparty computation, definitions

[*]Boston University and Tel-Aviv University. Email: `canetti@tau.ac.il`. Supported by the Check Point Institute for Information Security, an NSF EAGER grant, and NSF Algorithmic Foundations grant no. 1218461.

[†]Department of Computer Science, Bar-Ilan University. Email: `asafc@me.com`. Work supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 189/11).

[‡]Department of Computer Science, Bar-Ilan University. Email: `lindell@biu.ac.il`. Work supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 189/11).

1 Introduction

1.1 Background

The framework of universally composable security (UC) provides very strong security guarantees. In particular, a protocol that is UC secure maintains its security properties when run together with many other arbitrary secure and insecure protocols. To be a little more exact, if a protocol π UC securely realizes some ideal functionality $\mathcal{F}$, then π will "behave just like $\mathcal{F}$" in whatever arbitrary computational environment it is run. This security notion matches today's computational and network settings and thus has become the security definition of choice in many cases.

One of the strengths of the UC framework is that it is possible to express almost any cryptographic task as a UC ideal functionality, and it is possible to express almost any network environment within the UC framework (e.g., authenticated and unauthenticated channels, synchronous and asynchronous message delivery, fair and unfair protocol termination, and so on). However, this generality and power of expression comes at the price of the UC formalization being very complicated. It is important to note that many of these complications exist in order to enable general cryptographic tasks to be expressible within the framework. For example digital signatures involve local computation alone, and also have no a priori polynomial bound on how many signatures will be generated (by an honest party) since the adversary can determine this. This is very different from standard "secure computation tasks" that involve an a priori known number of interactions between the honest parties.

In this paper, we present a simpler and more restricted variant of the universally composable security (UC) framework; we call this framework simple UC, or SUC for short. Our simplified framework suffices for capturing classic secure computation tasks like secure function evaluation, mental poker, and the like. However, it does not capture more general tasks like digital signatures, and has a more rigid network model (e.g., the set of parties is a priori fixed and authenticated channels are built into the framework). These restrictions make the formalization much simpler, and far closer to the classic stand-alone definition of security which many are more familiar with. Importantly, our simplifications are with respect to the *expressibility* of the framework and *not the security guarantees obtained*. Thus, we can prove that any protocol that is expressed and proven secure in the SUC framework can be automatically transformed into a protocol that is secure also in the full UC framework (relative to an appropriately modified ideal functionality). This means that it is possible to work in the simpler SUC framework, and automatically obtain security in the full UC framework.

Remark: We assume familiarity with the ideal/real model paradigm and the standard definitions of security for multiparty computation; see [3] and [14, Chapter 7] for a detailed treatment and discussion on these definitions. In addition, we assume that the reader has some basic familiarity and understanding of the notion of universally composable security. This paper is not intended as a tutorial of universally composable security.

1.2 An Informal Introduction to Universally Composable Security

We begin by informally outlining the framework for universally composable security [4, 7]. The framework provides a rigorous method for defining the security of cryptographic tasks, while ensuring that security is maintained under concurrent general composition. This means that the protocol remains secure when run concurrently with arbitrary other secure and insecure protocols. Protocols that fulfill the definitions of security in this framework are called universally composable (UC).

As in other general definitions (e.g., [15, 24, 1, 25, 3, 14]), 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 (in one or more iterations). We call the algorithm run by the trusted party an ideal functionality. Since the trusted party just runs the ideal functionality, we do not distinguish between them. Rather, we refer to *interaction between the parties and the functionality.* Informally, a protocol securely carries out a given task if no adversary can gain more from an attack on a real execution of the protocol, than from an attack on an ideal process where the parties merely hand their inputs to a trusted party with the appropriate functionality and obtain their outputs from it, without any other interaction. In other words, it is required that a real execution can be *emulated* in the above ideal process (where the meaning of *emulation* is described below). We stress that in a real execution of the protocol, no trusted party exists and the parties interact amongst themselves.

In order to prove the universal composition theorem, the notion of emulation in the UC framework is considerably stronger than in previous ones. Traditionally, the model of computation includes the parties running the protocol, plus an adversary $\mathcal{A}$ that potentially corrupts some of the parties. In the setting of concurrency, the adversary also has full control over the scheduling of messages (i.e., it fully determines the order that messages sent between honest parties are received); thus, the model is inherently asynchronous. Emulation means that for any adversary $\mathcal{A}$ attacking a real protocol execution, there should exist an "ideal process adversary" or simulator $\mathcal{S}$, that causes the outputs of the parties in the ideal process to be essentially the same as the outputs of the parties in a real execution. In the universally composable 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. (As is hinted by its name, $\mathcal{Z}$ represents the external environment that consists of arbitrary protocol executions that may be running concurrently with the given protocol.) A protocol is said to UC-securely compute a given ideal functionality $\mathcal{F}$ if for any "real-life" adversary $\mathcal{A}$ that interacts with the protocol 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 $\mathcal{F}$.) Note that the definition requires the "ideal-process adversary" (or simulator) $\mathcal{S}$ to interact with $\mathcal{Z}$ throughout the computation. Furthermore, $\mathcal{Z}$ cannot be "rewound".

The following *universal composition theorem* is proven in [4, 7]: Consider a protocol π that operates in a *hybrid* model of computation where parties can communicate as usual, and in addition have ideal access to an unbounded number of copies of some ideal functionality $\mathcal{F}$. (This model is called the $\mathcal{F}$-hybrid model.) Furthermore, let ρ be a protocol that UC-securely computes $\mathcal{F}$ as sketched above, and let π^ρ be the "composed protocol". That is, π^ρ is identical to ρ with the exception that each interaction with the ideal functionality $\mathcal{F}$ is replaced with a call to (or an activation of) an appropriate instance of the protocol ρ. Similarly, ρ-outputs are treated as values provided by the functionality $\mathcal{F}$. The theorem states that in such a case, π and π^ρ have essentially the same input/output behaviour. Thus, ρ behaves just like the ideal functionality $\mathcal{F}$, even when composed concurrently with an arbitrary protocol π. This implies the notion of concurrent general composition. A special case of the composition theorem states that if π UC-securely computes some ideal functionality $\mathcal{G}$ in the $\mathcal{F}$-hybrid model, then π^ρ UC-securely computes $\mathcal{G}$ from scratch.

In order to model dynamic settings, the UC formulation enables programs to dynamically generate other programs and dynamically determine their code, and a control function must be defined to determine what operations are allowed and not allowed. This model provides great flexibility, and enables one to model almost any conceivable setting. However, this also adds considerable complexity to the definition, in part due to subtleties that arise with respect to polynomial time, and with respect to the communication rules [17, 18, 21].

1.3 The SUC Framework

The SUC framework is designed to be as similar as possible to the stand-alone definitions of secure multiparty computation (cf. [3, 14]), with the addition of an interactive environment as is required for proving concurrent general composition [22]. In this section we outline the SUC definition, and discuss the main differences between it and the full UC framework.

1.3.1 An Outline of the SUC Framework

The SUC framework was designed by starting with the stand-alone model of secure computation, and adding the seemingly minimal changes required to obtain security under concurrent general composition for standard secure computation tasks, without many of the complications of the UC framework. Thus, in the SUC framework a *fixed* set of parties interact with each other and/or with an ideal functionality (depending on whether an execution is real, ideal or hybrid). An adversary may corrupt some subset of the parties, in which case it sees their state and controls them in the standard way depending on whether it is semi-honest or malicious. As in the UC framework, an environment machine $\mathcal{Z}$ interacts with the adversary throughout the computation and serves as an "interactive distinguisher" between a real execution of the protocol and an ideal execution.

In order to model the fact that the adversary controls all message scheduling, the parties (and any ideal functionality) are connected in a *star configuration* via a router machine. The router queues all communication, and forwards messages only when instructed by the adversary. The adversary sees all the messages sent, and delivers or blocks these messages at will. We note that although the adversary may block messages, it cannot modify messages sent by honest parties (i.e., the communication lines are ideally authenticated). Thus messages sent by a party can arrive in a different order or not arrive at all, but cannot be forged unless the adversary has corrupted the sending party. In order to model the fact that inputs sent to ideal functionalities are private, the SUC framework defines that any message between the parties and the ideal functionality is comprised of a public header and private content. The public header contains any information that is public and thus revealed to the adversary (e.g., the type of message is being sent or what its length is), whereas the private content contains information that the adversary is not supposed to learn.

Composition is defined by replacing the Turing machine code for sending a message to an ideal functionality by the Turing machine code of the protocol that realizes the functionality. Thus, subroutines are executed internally as in the sequential modular composition modeling in [3], and unlike the modeling in the full UC framework where subprotocols are invoked as separate ITMs.

1.3.2 The Main Differences

Defining polynomial time. In the UC framework, machines can be dynamically added to the computation through the mechanism of an external write instruction. Thus, bounding the running time of a single machine by a polynomial does not guarantee that the overall computation is bounded by polynomial time. For example, consider an execution with a single machine that generates a copy of itself and halts. Clearly, each machine is polynomial time. However, an execution of this machine will generate an infinite series of machines and will thus never halt. This makes defining polynomial time in this setting difficult. The definition in the UC framework states that a machine M runs in polynomial time if it runs at most $p(\tilde{n})$ steps where p is a polynomial, and $\tilde{n}$ is the length of the input tape of M plus the security parameter, *minus* the length of all the inputs M provides to other machines. It can be shown that under this definition, the overall execution is bounded by a polynomial, and pathological examples like the one provided above are ruled out.

In the SUC framework, machines cannot generate other machines, and the set of all machines running is fixed ahead of time. Thus, the aforementioned challenges do not arise. We can therefore define polynomial time in the more standard way by simply requiring that each machine run in $p(|x| + n)$ steps, where $|x|$ is the length of its input and n is the security parameter.

Authentication versus unauthenticated channels. The basic UC framework has plain, unauthenticated channels; authenticated channels are obtained via an ideal functionality $\mathcal{F}_{\text{AUTH}}$ that provides message authentication. However, almost all secure computation protocols rely on authenticated channels and this is the modeling used in [3, 14]. We therefore adopt authenticated channels as the default in the SUC framework, thus simplifying the description of protocols. Another way of saying this is that the real model of computation in the SUC framework corresponds to the $\mathcal{F}_{\text{AUTH}}$-hybrid model of computation in the UC framework. Although this is mainly an **aesthetic** difference, it makes protocol descriptions much more simple.

Defining composition. The dynamic generation of machines in the UC framework also adds complications regarding defining composition. For example, security under composition is only guaranteed to hold for *subroutine respecting protocols*, which places limitations on the input/output interface of machines with other machines; see [4, Section 5.1]. These difficulties arise since when a party calls a subroutine in the UC framework, the subroutine machine is a distinct machine. In order to simplify this issue, in the SUC framework a subroutine call is simply a call to a local routine on the same machine, *exactly* as in the formulation of sequential modular composition in [3].

1.3.3 The UC Security of SUC Protocols

We define a transformation $T_P : SUC \to UC$ that translates SUC-protocols to UC-protocols, and a transformation ϕ that translates ideal functionalities from the SUC framework to the UC framework. We prove that a protocol π SUC-securely computes some ideal functionality $\mathcal{F}$ if and only if $T_P(\pi)$ UC-securely computes $\phi(\mathcal{F})$. SUC composition is derived as a result.

The implication is that one may build secure computation protocols in SUC and automatically derive UC security without working with the complex structures of the UC framework. Composition of SUC and UC protocols can also be done freely.

Since SUC is less expressive than UC, it is not possible to express every functionality in SUC. SUC cannot replace UC, but is intended as a convenient "API" to the UC framework that offers the same security standard, and can simplify the process of proving UC security of protocols.

1.4 Related Work

There has been considerable work in refining the UC framework and solving all the subtleties that arise in the fully dynamic and concurrent setting [17, 19, 16]. In addition, there have been other frameworks developed to capture the same setting of dynamic concurrency as that of the UC framework. These include [23, 20, 18, 21, 25]. However, all of these attempt to capture the same generality of the UC framework in alternative ways. In this work, we make no such attempt. Rather, our aim is to capture concurrency for more restricted tasks, and thus obtain a simpler definition.

2 The Simpler UC Model and Definition

In this section, we present a simpler variant of universally composable security that is suitable for standard multiparty computation tasks. It does not have the generality and expressibility of the full-fledged UC framework, but suffices for classic secure computation tasks where a set of parties compute some function of their inputs (a.k.a. secure function evaluation). It also suffices for reactive computations where parties give inputs and get outputs in stages.

2.1 Preliminaries

We denote the security parameter by n. A function $\mu : \mathbb{N} \to [0,1]$ is negligible if for every polynomial $p(\cdot)$ there exists a value $n_0 \in \mathbb{N}$ such that for every $n > n_0$ it holds that $\mu(n) < 1/p(n)$. All entities (parties, adversary, etc.) are interactive Turing machines (ITM); each such machine has an input tape, an output tape, an incoming communication tape, an outgoing communication tape, and a security parameter tape. If the machine is probabilistic then it also has a random tape. The value written on the security parameter tape is in unary.

We say that a machine is polynomial time if it runs in time that is polynomial in the sum of the lengths of the values that are written on its input tape during its execution plus the security parameter (note that in reactive computations there may be many inputs). Thus, we require that there exists a polynomial $q(\cdot)$ so that for any series of inputs $x_1, x_2, ..., x_\ell$ written on the machine's input tape throughout its lifetime, it always halts after at most $q(n + |x_1| + |x_2| + \cdots + |x_\ell|) = q\left(n + \sum_{j=1}^{\ell} |x_j|\right)$ steps. This is equivalent to saying that each machine receives 1^n as its first input, and n is polynomial in the sum of the lengths of all of its inputs.

It is important to note that even if the inputs are short (e.g., constant length), a polynomial-time party can still run in time that is polynomial in the security parameter in every invocation. In order to see this, observe that $\left(n + \sum_{j=1}^{\ell} |x_j|\right)^2 > \sum_{j=1}^{\ell} n \cdot |x_j|$ and thus a machine that runs in time n^c in every invocation is polynomial-time by taking $q(n + \sum_{j=1}^{\ell} |x_j|) = (n + \sum_{j=1}^{\ell} |x_j|)^{2c}$.

2.2 Interactive Turing Machines

We treat ITMs in the SUC model as universal Turing machines. We assume all ITMs have the same state function δ of the universal Turing machine.

Definition 2.1 *An* Interactive Turing Machine in the SUC environment *M is a universal Turing machine with the following tapes:*

Special tapes:

- *A read-only* code tape. *This tape contains a description, using some standard encoding, of the program of M. This description is readable by the state function δ of the universal Turing machine, and is called the code of M. The code of M must also contain a unique integer i called the identifier of M (or identity of M); for identifier i we call M the machine M_i. In the context of protocol parties, the identifier of the party machine is a* unique party identifier *(or* pid*).*

- *A read only dedicated* input tape*.*
- *A write only dedicated* output tape*. This tape is also "write-once", in the sense that once M writes a value on a cell of this tape, M cannot write again on the same cell.*
- *A read only* security parameter tape *containing the security parameter in unary representation.*
- *A read only* incoming communication tape*. By convention, data arriving on this tape always begins with a string $\sigma = (S, R)$ where S is the identity of the machine the data originated from, and R is the identity of machine M. M might have several such tapes (e.g., the adversary and router have more than one; see Figure 1).*
- *A write only* outgoing communication tape*. By convention, each separate segment of data written on this tape is prefixed with a string $\sigma = (S, R)$ where S is the identity of machine M, and R is a different identity. M might have several such tapes.*
- *A read only* random tape *containing an infinite sequence of random bits.*
- *A read-write* work tape*.*

In addition, M has the following special features:

- *A special* read next-message *instruction. This instruction specifies either the* input *or the* incoming communication *tape. The effect of this instruction, is that the reading head jumps to the beginning of the next message in a single step.*[1]
- *A special* HALT *input. M may receive a special input denoted as* HALT *on the input tape; upon receiving such input M halts immediately.*

If M is an Interactive Turing Machine in the SUC environment, we denote $M \in ITM^{SUC}$. If M also runs in polynomial time (as defined in Section 2.1) we denote $M \in PPT^{SUC}$.

In the next section, we will define what it means for a set of ITMs to interact in the SUC framework. Machines interact via shared tapes. That is, when a machine M_1 is activated and writes on its outgoing communication tape, this tape is also the incoming communication tape of another machine M_2. When M_1 finishes writing the relevant data for the next machine to run, its activation ends and the next machine is activated according to the rules described in Section 2.3.

We assume familiarity with the definition of an ITM in the full UC framework (see [4, 7]). Such ITMs are referred in this paper as UC Interactive Turing Machines, and we denote them by $M' \in ITM^{UC}$ and $M' \in PPT^{UC}$.

2.3 The Communication and Execution Models

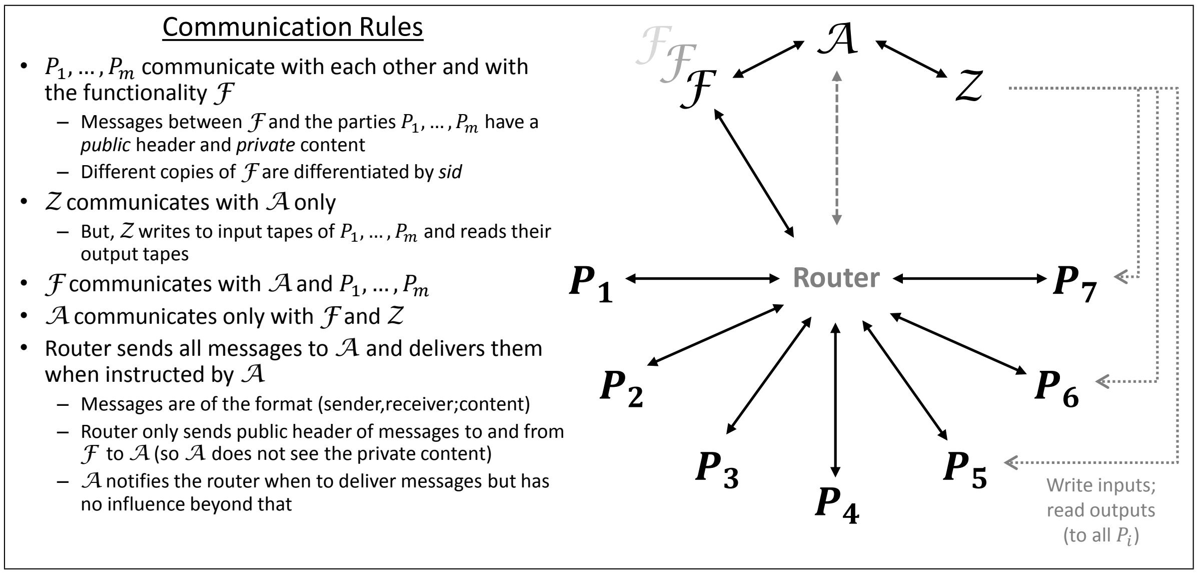
We consider a network where the adversary sees all the messages sent, and delivers or blocks these messages at will. We note that although the adversary may block messages, it cannot modify messages sent by honest parties (i.e., the communication lines are ideally authenticated). We

[1]To implement this instruction, we assume that each message or input ends with a special end-of-message (eom) character.

consider a completely asynchronous point-to-point network, and thus the adversary has full control over when messages are delivered, if at all.

We now formally specify the communication and execution model. This general model is the same for the real, ideal and hybrid models; we will describe below how each of the specific models are derived from the general communication and execution model.

Communication. In each execution there is an environment $\mathcal{Z}$, an adversary $\mathcal{A}$, participating parties $P_1, \ldots, P_m$, and possibly an ideal functionality $\mathcal{F}$. The parties, adversary and functionality are "connected" in a star configuration, where all communication is via an additional *router machine* that takes instructions from the adversary (see Figure 1). Formally, this means that the outgoing communication tape of each machine is connected to the incoming communication tape of the router, and the incoming communication tape of each machine is connected to the outgoing communication tape of the router. (For this to work, we define the router so that it has one incoming and one outgoing tape for every other entity in the network except the environment). As we have mentioned, the adversary has full control over the scheduling of all message delivery. Thus, whenever the router receives a message from a party it stores the message and forwards it to the adversary $\mathcal{A}$. Then, whenever the adversary wishes to deliver a message, it sends it to the router who then checks that this message has been stored. If yes, it delivers the message to the designated recipient and erases it, thereby ensuring that every message is delivered only once. If no, the router just ignores the message. If the same message is sent more than once, then the router will store multiple copies and will erase one every time it is delivered.



Communication Rules

- $P_1, \ldots, P_m$ communicate with each other and with the functionality $\mathcal{F}$
  - Messages between $\mathcal{F}$ and the parties $P_1, \ldots, P_m$ have a *public* header and *private* content
  - Different copies of $\mathcal{F}$ are differentiated by *sid*
- $\mathcal{Z}$ communicates with $\mathcal{A}$ only
  - But, $\mathcal{Z}$ writes to input tapes of $P_1, \ldots, P_m$ and reads their output tapes
- $\mathcal{F}$ communicates with $\mathcal{A}$ and $P_1, \ldots, P_m$
- $\mathcal{A}$ communicates only with $\mathcal{F}$ and $\mathcal{Z}$
- Router sends all messages to $\mathcal{A}$ and delivers them when instructed by $\mathcal{A}$
  - Messages are of the format (sender,receiver;content)
  - Router only sends public header of messages to and from $\mathcal{F}$ to $\mathcal{A}$ (so $\mathcal{A}$ does not see the private content)
  - $\mathcal{A}$ notifies the router when to deliver messages but has no influence beyond that

Figure 1: The communication model and rules

Observe that $\mathcal{A}$ can only influence when a message is delivered but cannot modify its content. This therefore models authenticated channels, which is standard for secure computation. By convention, a message x from a party P_i to P_j will be of the form (P_i, P_j, x); after P_i writes this message to its outgoing communication tape, the router receives it and checks that the correct sending party identifier P_i is written in the message; if yes, it stores it and works as above (sending only to the P_j designated in the message); if no, it ignores the message. Observe that this means that P_j also knows who sent the message to it.

In addition, we assume that the set of parties is fixed ahead of time and known to all.[2]

The above communication model is the same regarding the communication between the functionality $\mathcal{F}$ and the parties and adversary, with two differences. First, the different copies of $\mathcal{F}$ are differentiated by a unique session identifier sid for each copy. Specifically, each message sent to the ideal functionality has a session identifier sid. When, the "main ideal functionality" receives a message, it first checks if there exists a copy of the ideal functionality with that sid. If not, then it begins a new execution of the actual ideal functionality code with that sid, and executes the functionality on the given message. If a copy with that sid does already exist, then that copy is invoked with the message. Likewise, any message sent from a copy of the ideal functionality to a party is sent together with the sid identifying that copy.

The second difference is that any message between the parties and the ideal functionality is comprised of a public header and private content. The public header contains any information that is public and thus revealed to the adversary, whereas the private content contains information that the adversary is not supposed to learn. For example, in a standard two-party computation functionality where $\mathcal{F}$ computes $f(x, y)$ for some function f (where x is P_1's input and y is P_2's input), the inputs x and y sent by the parties to $\mathcal{F}$ are private. The output from $\mathcal{F}$ to the parties may be public or private, depending on whether this output is supposed to remain secret (say from an eavesdropping adversary between two honest parties) even after the computation.[3] A more interesting example is the commitment functionality, in which the public header would also contain the message type (i.e., "commit" or "reveal"), since we typically do not try to hide whether the parties are running a commitment or decommitment protocol. Formally, upon receiving a message from a participating party P_i for the functionality or vice versa, the router forwards only the sender/receiver identities and the *public header* to the adversary; the private content is simply not sent.[4]

We stress that in the SUC framework, the adversary determines when to deliver a message from $\mathcal{F}$ to participating parties $P_1, \ldots, P_m$ in the same way as between two participating parties. This is unlike the UC framework where the adversary has no such power. In the UC model ideal functionalities are invoked as subroutine machines, and the protocol parties of the *main instance* communicate with the invoked *sub-protocol machine* directly via the input and output tapes, without passing through the adversary. Thus, in the class of functionalities that can be expressed in SUC is more restricted. Specifically, we cannot guarantee fairness in the SUC framework, nor model local computation via an ideal functionality (as is used to model digital signatures and public-key encryption in the UC framework).

Finally, the environment $\mathcal{Z}$ communicates with the adversary directly and not via the router. This is due to the fact that it cannot send messages to anyone apart from the adversary; this includes the ideal functionality $\mathcal{F}$. However, differently to all other interaction between parties, the environment $\mathcal{Z}$ can write inputs to the honest parties' input tapes and can read their output tapes (we do not call this "communication" in the same sense since it is not via the communication tapes). The adversary $\mathcal{A}$ itself can send messages in the name of any corrupted party (see Section 2.4 below), and can send messages to $\mathcal{Z}$ and $\mathcal{F}$ (the fact that it *can* communicate with $\mathcal{F}$ is useful for relaxing functionalities to allow some adversarial influence; see [4, 7]). The adversary $\mathcal{A}$ cannot

[2]Observe that in contrast to the full UC model, a protocol party here cannot write to the input tapes of other parties. All communication between protocol parties is via the router.

[3]If one of the parties is corrupted then $f(x, y)$ is always learned by the adversary. However, if both are honest, then it may or may not be learned depending on how one defines it.

[4]In order to formalize this, every ideal functionality $\mathcal{F}$ has an associated public-header function $H_{\mathcal{F}}(x)$ that defines the public-header portion of the input x.

"directly" communicate with the participating parties.

Execution. An execution of a set of machines connected as above and communicating according to the above rules proceeds as follows. First, all machines are initialized to have the same value 1^n on their security parameter tapes. Second, the environment is given an initial input $z \in \{0,1\}^*$ and is the first to be "activated".

In the concurrent setting, and unlike the classic stand-alone setting for secure computation, there are no synchronous rounds in which all parties send messages, compute their next message, and then send it. Rather, the adversary is given full control over the scheduling of messages sent. In order to model this but still to have a well-defined execution model, an execution is modeled by a series of activations of machines one after another, where the order of activations is determined by the adversary. As we have stated, the environment $\mathcal{Z}$ is activated first. In any activation of the environment, it may write to the input tapes of any of the participating parties $P_1, \ldots, P_m$ that it wishes to, and read their output tapes. In addition, it can send a message to the adversary by writing on its outgoing communication tape. When it halts, the adversary is activated next. In any activation of the adversary, it may read all messages written to entities' outgoing communication tapes (apart from the private content sent between a party and $\mathcal{F}$), carry out any local computation, and write a message on its outgoing communication tape to $\mathcal{Z}$. It then completes its activation by doing one of the following:

1. Instructing the router to deliver a message to any single party that it wishes (including messages between the parties and $\mathcal{F}$). In this case the router is activated next to deliver the message. After the router has delivered the message the recipient party (or $\mathcal{F}$) is activated.

2. Sending a direct message to $\mathcal{F}$ (this type of communication is not via the router). In this case $\mathcal{F}$ is activated next.

3. Sending a direct message to $\mathcal{Z}$. In this case $\mathcal{Z}$ is activated next.

If the activated machine is $\mathcal{F}$ or $\mathcal{Z}$, it reads the message from $\mathcal{A}$, runs a local computation and then sends a response to $\mathcal{A}$, in which case $\mathcal{A}$ is activated next. Otherwise, the activated party ($P_1, \ldots, P_m$ or $\mathcal{F}$) can read the message on its incoming communication tape, carry out any local computation it wishes, and write any number of messages to its outgoing communication tape to the router; its activation ends when it halts. The router is activated next and sends all of the messages that it received to $\mathcal{A}$. The adversary is then once again activated, and so on. One technicality is that the adversary may wish to activate a party to whom no message has previously been sent. This makes most sense at the beginning of a protocol execution where a party already has input but has not yet been sent any messages. Since the adversary is not generally allowed to communicate to parties, it cannot activate such a party since there are no messages to deliver. We therefore allow the adversary to deliver an "empty message" to a party to activate it whenever it wishes. The execution ends when the environment writes a bit to its output tape (the fact that the environment's output is just a single bit is without loss of generality, as shown in [4, 7]).

We stress that the ideal functionality has no input on its input tape and never writes to its output tape; it only communicates with the participating parties and the adversary.

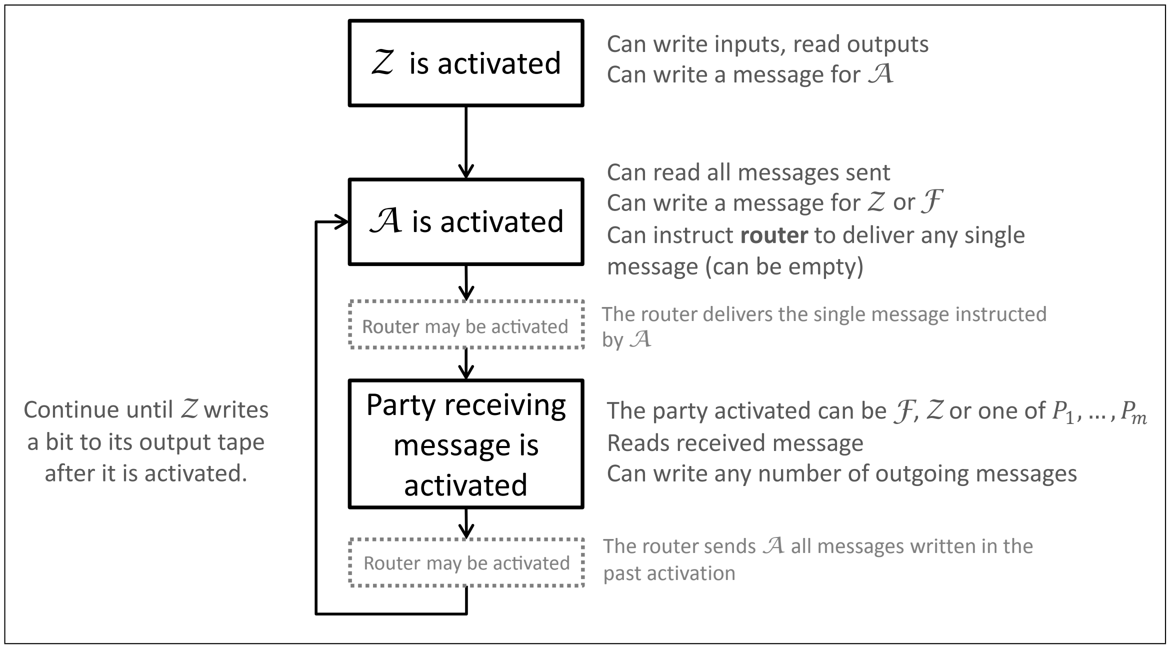

$\mathcal{Z}$ is activated
Can write inputs, read outputs
Can write a message for $\mathcal{A}$
$\mathcal{A}$ is activated
Can read all messages sent
Can write a message for $\mathcal{Z}$ or $\mathcal{F}$
Can instruct **router** to deliver any single message (can be empty)
Router may be activated
The router delivers the single message instructed by $\mathcal{A}$
Party receiving message is activated
The party activated can be $\mathcal{F}$, $\mathcal{Z}$ or one of $P_1, \ldots, P_m$
Reads received message
Can write any number of outgoing messages
Router may be activated
The router sends $\mathcal{A}$ all messages written in the past activation
Continue until $\mathcal{Z}$ writes a bit to its output tape after it is activated.

Figure 2: The execution flow and order of activations

2.4 Corruptions and Adversarial Power

As in the standard model of secure computation, the adversary is allowed to corrupt parties. In the case of static adversaries the set of corrupted parties is fixed at the onset of the computation. In the adaptive case the adversary corrupts parties at will throughout the computation. In the static corruption case, the environment $\mathcal{Z}$ is given the set of corrupted parties at the onset of the computation. In the active corruption case, whenever the adversary corrupts a party, $\mathcal{Z}$ is notified of the corruption immediately. The adversary is allowed to corrupt parties whenever it is activated. (Formally, the adversary sends a $(\mathsf{corrupt}, P_i)$ message first to P_i via the router, and P_i returns its full internal state to the adversary. Then, the adversary sends the corrupt message to $\mathcal{Z}$ who is activated at the end of the corruption sequence.)

We also distinguish between malicious and semi-honest adversaries: If the adversary is malicious then corrupted parties follow the arbitrary instructions of the adversary. In the semi-honest case, even corrupted parties follow the prescribed protocol and the adversary only gets read access to the internal state of the corrupted parties. In the case of a *malicious* adversary, we stress that the adversary can send any message that it wishes in the name of a corrupted party. Formally, this means that the router delivers any message in the name of a corrupted party at the request of the adversary. Observe that in the case of *adaptive malicious corruptions*, any messages that were sent by a party (to another party or to the ideal functionality) before it was corrupted but were not yet delivered may be modified arbitrarily by the adversary. This follows from the fact that from the point of corruption the router delivers any message requested by the adversary. This mechanism assumes that the router is notified whenever a party is corrupted.

We stress that unlike in the full UC model, here it is not possible to "partially corrupt" a party. Rather, if a party is corrupted, then the adversary learns everything. This means that we cannot model, for example, the *forward security* property of key exchange that states that if a party's session key is stolen in one session, then this leaks nothing about its session key in a different session (since modeling this requires corrupting one session of the key exchange and not another). Also, *proactive security* cannot be modelled in SUC [10].

2.5 The Real, Ideal and Hybrid Models

We are now ready to define the real, ideal and hybrid models. These are all just special cases of the above communication and execution models:

- The real model with protocol π: In the real model, there is no ideal functionality and the (honest) parties send messages to each other according to the specified protocol π. We denote the output bit of the environment $\mathcal{Z}$ after a real execution of a protocol π with environment $\mathcal{Z}$ and adversary $\mathcal{A}$ by $\textsc{suc-real}_{\pi,\mathcal{A},\mathcal{Z}}(n,z)$, where z is the input to $\mathcal{Z}$ and n is the security parameter.

- The ideal model with $\mathcal{F}$: In the ideal model with $\mathcal{F}$ the parties follow a *fixed ideal-model protocol*. According to this protocol, the parties send messages only to the ideal functionality but never to each other. Furthermore, these messages are the inputs that they read from their input tapes, and nothing else (unless they are corrupted and the adversary is malicious, in which case they can send anything to $\mathcal{F}$). In addition, they write any message received back from the ideal functionality to their output tapes. That is, the ideal-model protocol instructs a party upon activation to read any new input on its input tape and send it unmodified to $\mathcal{F}$ as an outgoing message, and to read all incoming messages (from $\mathcal{F}$) on its incoming message tape and write them unmodified to its output tape. This then ends the party's activation. We denote the output of $\mathcal{Z}$ after an ideal execution with ideal functionality $\mathcal{F}$ and adversary $\mathcal{S}$ (denoted by $\mathcal{S}$ since it is actually a "simulator") by $\textsc{suc-ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(n,z)$, where n and z are as above. We stress that in the ideal model, the adversary/simulator $\mathcal{S}$ interacts with $\mathcal{Z}$ in an online way; in particular, it cannot rewind $\mathcal{Z}$ or look at its internal state. In addition, in keeping with the general communication model all messages between the parties and $\mathcal{F}$ are delivered by the adversary.[5]

- The hybrid model with π and $\mathcal{F}$: In the hybrid model, the parties follow the protocol π as in the real model. However, in addition to regular messages sent to other parties, π can instruct the parties to send messages to the ideal functionality $\mathcal{F}$ and also instructs them how to process messages received from $\mathcal{F}$. We stress that the messages sent to $\mathcal{F}$ may be any values specified by π and are not limited to inputs like in the ideal model. We denote the output of $\mathcal{Z}$ from a hybrid execution of π with ideal calls to $\mathcal{F}$ by $\textsc{suc-hybrid}^{\mathcal{F}}_{\pi,\mathcal{A},\mathcal{Z}}(n,z)$, where $\mathcal{A},\mathcal{Z},n,z$ are as above. When $\mathcal{F}$ is the ideal functionality we call this the $\mathcal{F}$-hybrid model.

In all models, there is a fixed set of participating parties $P_1,\ldots,P_m$, where each party has a unique party identifier. Observe that we formally consider a *single* ideal-functionality type $\mathcal{F}$, and not multiple different ones.[6] This is not a limitation even though protocols often use multiple different subprotocols (e.g., commitment, zero knowledge, and oblivious transfer). This is because one can

[5]The fact that the adversary delivers these messages and thus message delivery is not guaranteed frees us from the need to explicitly deal with the "early stopping" problem of protocols run between two parties or amongst many parties where only a minority may be honest. This is because the adversary can choose which parties receive output and which do not, even in the ideal model.

[6]This is not to be confused with multiple copies of the same functionality $\mathcal{F}$ which is included in the model.

define a single functionality computing multiple subfunctionalities. Thus, formally we consider one. When defining protocols and proving security, it is customary to refer to multiple functionalities with the understanding that this is formally taken care of as described.

2.6 The Definition and Composition Theorem

We are now ready to define SUC security, and to state the composition theorem. Informally, security is defined as in the classic stand-alone definition of security by requiring the existence of an ideal-model simulator for every real-model adversary. However, in addition, the simulator must work for every environment, as in the aforementioned communication and execution models. The environment behaves as the interactive distinguisher, and therefore we say that a protocol π SUC-securely computes a functionality if the environment outputs 1 with almost the same probability in a real execution of π with $\mathcal{A}$ as in an ideal execution with $\mathcal{F}$ and $\mathcal{S}$. Recall that the SUC-IDEAL and SUC-REAL notation denotes the output of $\mathcal{Z}$ after the respective executions.

Balanced Environments. A balanced environment is an environment for which at any point in time during the execution, the overall length of the inputs given to the parties of the main instance of the protocol is at most n times the length of the input to the adversary [7]. As in the full UC framework, we require balanced environments in order to prevent unnatural situations where the input length and communication complexity of the protocol is arbitrarily large relative to the input length and complexity of the adversary. In such case no PPT adversary can deliver even a fraction of the protocol communication. The definition of UC security considers only balanced environments, and we adopt this same convention.

Definition 2.2 *Let π be a protocol for up to m parties and let $\mathcal{F}$ be an ideal functionality. We say that π SUC-securely computes $\mathcal{F}$ if for every probabilistic polynomial-time real-model adversary $\mathcal{A}$ there exists a probabilistic polynomial-time ideal-model adversary $\mathcal{S}$ such that for every probabilistic polynomial-time balanced environment $\mathcal{Z}$ and every constant $d \in \mathbb{N}$, there exists a negligible function $\mu(\cdot)$ such that for every $n \in \mathbb{N}$ and every $z \in \{0,1\}^*$ of length at most n^d,*

$$\left|\Pr[\textsc{suc-ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(n,z) = 1] - \Pr[\textsc{suc-real}_{\pi,\mathcal{A},\mathcal{Z}}(n,z) = 1]\right| \leq \mu(n).$$

We remark that although $\mathcal{Z}$ is a uniform machine, since it receives an arbitrary z as input this essentially makes it non-uniform (z can be viewed as advice and can be different for every n).

We now conclude by stating the composition theorem. Before formally stating it, we need to define what it means to replace the ideal calls to $\mathcal{F}$ in an $\mathcal{F}$-hybrid protocol π with a subprotocol execution of some protocol ρ. Let π be a protocol for the $\mathcal{F}$-hybrid model, and let ρ be a protocol (that itself may be designed in a hybrid model for some other ideal functionality $\mathcal{G}$). We define the protocol π^ρ as in [3] as follows: whenever π instructs a party to send a message (x) to $\mathcal{F}$ with identifier sid, the party in π^ρ begins an execution of ρ with input (x) and identifier sid. All messages received by the party that are intended for the copy of ρ with identifier sid are internally routed by the party to the appropriate copy of ρ. Finally, the output of the execution of ρ with identifier sid is interpreted by the party as the message received back from $\mathcal{F}$ with identifier sid from the call in π. Formally, the above is defined by replacing the Turing machine code for sending a message to $\mathcal{F}$ by the Turing machine code of ρ, and so on. Thus, all Turing machines π, ρ and π^ρ are standard Interactive Turing machines as defined above and π^ρ is derived from π by adding the code of ρ in place of messages sent to $\mathcal{F}$. This is different than UC composition, where the subprotocol is invoked as separate ITMs. Keeping a fixed number of ITMs throughout the computation simplifies the model and allows us to use the simpler polynomial time definition.

In order for the composition theorem to hold, we need to make one limitation on the use of ideal calls. Specifically, let ρ be a protocol that securely computes some $\mathcal{F}$ but is itself in the hybrid model with some ideal functionality $\mathcal{G}$ (e.g., ρ may be a zero-knowledge protocol that uses a commitment functionality, or a commitment protocol that uses a common reference string functionality). Then, we require that two different instances of protocol ρ call different copies of $\mathcal{G}$. That is, the different instances of ρ must use different session identifiers for their calls to $\mathcal{G}$. This can be enforced by a simple naming convention, as follows. The main protocol is given a session identifier sid. Then, each subprotocol of the protocol is given session identifier $\mathsf{sid}\|\mathsf{ssid}$, where ssid is a unique subsession identifier under sid. Thus, if a protocol π with identifier sid uses two calls to an ideal $\mathcal{F}$, then the session identifiers of $\mathcal{F}$ will be some $\mathsf{sid}\|\mathsf{ssid}_1$ and $\mathsf{sid}\|\mathsf{ssid}_2$ (with $\mathsf{ssid}_1 \neq \mathsf{ssid}_2$). Now, when replacing $\mathcal{F}$ with a subprotocol ρ that contains two calls to $\mathcal{G}$, we have that the identifiers of the four copies of $\mathcal{G}$ will be "$\mathsf{sid}\|\mathsf{ssid}_1\|\mathsf{ssid}'_1$", "$\mathsf{sid}\|\mathsf{ssid}_1\|\mathsf{ssid}'_2$","$\mathsf{sid}\|\mathsf{ssid}_2\|\mathsf{ssid}'_1$", and "$\mathsf{sid}\|\mathsf{ssid}_1\|\mathsf{ssid}'_2$". Observe that ρ always concatenates the same two session identifiers ssid'_1 and ssid'_2. However, since different instances of ρ have different session identifiers, and these are concatenated, this ensures that all calls to $\mathcal{G}$ have unique identifiers. We stress that ρ is not required to call two different copies of $\mathcal{G}$. Rather, what we require is that different copies of ρ call different copies of $\mathcal{G}$ (for however many copies each copy of ρ requires). When this requirement doesn't hold, the composition theorem below doesn't necessarily hold. However in that case the joint state composition theorem (JUC) may be applied [12].

The composition theorem states that if ρ SUC-securely computes $\mathcal{F}$, then π^ρ behaves just like π in the $\mathcal{F}$-hybrid model. In particular, $\mathcal{Z}$ cannot distinguish the cases.

Theorem 2.3 *Let π be a protocol for the $\mathcal{F}$-hybrid model, and let ρ be a protocol that SUC-securely computes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. Then, for every probabilistic polynomial-time real-model adversary $\mathcal{A}$ there exists a probabilistic polynomial-time ideal-model adversary $\mathcal{S}$ such that for every probabilistic polynomial-time environment $\mathcal{Z}$ there exists a negligible function $\mu(\cdot)$ such that for every $z \in \{0,1\}^*$ and every $n \in \mathbb{N}$,*

$$\left|\Pr[\textsc{suc-hybrid}^{\mathcal{G}}_{\pi^\rho,\mathcal{S},\mathcal{Z}}(n,z)=1] - \Pr[\textsc{suc-hybrid}^{\mathcal{F}}_{\pi,\mathcal{A},\mathcal{Z}}(n,z)=1]\right| \leq \mu(n).$$

An important special case of the above theorem is where π SUC-securely computes a functionality $\mathcal{H}$ in the $\mathcal{F}$-hybrid model. In this case, it follows that π^ρ SUC-securely computes $\mathcal{H}$ in the $\mathcal{G}$-hybrid model. Furthermore, if ρ SUC-securely computes $\mathcal{F}$ in the real model, then π^ρ SUC-securely computes $\mathcal{H}$ in the real model. That is:

Corollary 2.4 *Let π be a protocol that SUC-securely computes a functionality $\mathcal{H}$ in the $\mathcal{F}$-hybrid model. If protocol ρ SUC-securely computes $\mathcal{F}$ in the $\mathcal{G}$-hybrid (resp., real) model, then π^ρ SUC-securely computes $\mathcal{H}$ in the $\mathcal{G}$-hybrid (resp., real) model.*

Theorem 2.3 and Corollary 2.4 can be proven in an almost identical way to the proof of the composition theorem in [4, 7]. Alternatively, it can be proven via reduction to the full UC composition theorem. This is achieved by providing a mapping from SUC protocols and ideal functionalities to corresponding UC protocols and ideal functionalities, and showing that a protocol SUC-securely realizes an ideal functionality *if and only if* its corresponding UC protocol UC realizes the corresponding ideal functionality in the full UC model. This suffices since the protocol and functionality can be mapped to the UC model. Then, UC composition is applied, and finally the composed protocol is mapped back to the SUC model. We prove the first direction (that SUC security implies UC security of the corresponding protocol and ideal functionality) in Section 3; this direction is of independent interest since it shows that it suffices to prove security of a protocol in the SUC model in order to obtain security in the full UC model. The second direction (that UC security of the corresponding protocol and ideal functionality implies SUC security) is proven in Section 4.3.

We remark that it suffices in this work to prove Corollary 2.4, rather than the more general result of Theorem 2.3. This is due to the fact that Corollary 2.4 suffices for *constructing* protocols in a modular way. Once a protocol that securely computes some functionality in the SUC framework is obtained, then Theorem 3.13 (Section 3) can be used to conclude that it is secure in the full UC model as well. Once this is derived, the general composition result of Theorem 2.3 can be obtained from the full UC model itself.

3 Proving that SUC-Secure Protocols are UC Secure

In this section, we show that any protocol that is secure in the SUC model is also secure in the full UC model. Stating this formally is non-trivial since there are technical differences between the models, as described below. A priori, the UC adversary seems more powerful than the SUC adversary, but we will show that this is in fact not the case, and we can simulate any UC adversary via an SUC adversary. The formal theorem statement is given in Theorem 3.13 in Section 3.3. It states that an appropriately transformed SUC protocol UC realizes an appropriately transformed ideal functionality, where the main difference between the SUC ideal functionality and its transformed counterpart is related to the adversary's capability to determine when inputs are processed and when outputs are delivered. This capability is built into the SUC framework but not the UC framework; thus the capability must be added to the UC ideal functionality (via a mechanism similar to the delayed output methodology described in [7]).

3.1 Overview

We define a transformation T_P of protocols from SUC to UC and a transformation ϕ of ideal functionalities and prove that a protocol π SUC-securely computes the ideal functionality $\mathcal{F}$ (according to the definition here) if and only if the appropriately modified $T_P(\pi)$ UC-securely computes $\phi(\mathcal{F})$ under the full definition appearing in [4, 7]. This implies that we do not need to reprove the UC composition theorem, as it is implied by the proof in [4, 7]. In addition, protocols proven secure in this model can be easily incorporated into constructions in the full UC model, if desired. Of course, some adjustments to the protocol must be made in order to accommodate the differences between the models; these are made in the aforementioned transformation on the code of the protocol. Before the required adjustments are presented, we discuss the differences in detail.

Runtime complexity: The notion of polynomial-time in the full UC framework needs a way to ensure that it is not possible to invoke machines infinitely and still be considered "polynomial time". The p-bounded definition prevents this by regarding the bits on the input tape as "run tokens" (n_I), and making a machine pay n_O tokens every time it writes an input of length n_O to another machine. Each machine is allowed to run $p(\tilde{n})$ steps, where $p(\cdot)$ is a polynomial and

$\tilde{n} = n_I - n_O$ is the number of tokens that were not given to other machines. In [7] it is shown that this mechanism preserves a polynomial bound even with the ability to invoke other machines. It is not immediate that an SUC protocol that runs in polynomial time will run in polynomial time by the full UC definitions. Nonetheless, we will prove that is it indeed possible to transform an SUC protocol to a UC protocol and preserve polynomial time.

Authentication of messages: The SUC model assumes authenticated channels of communication between participating parties; this is not assumed in the full UC framework. The $\mathcal{F}_{\text{AUTH}}$ functionality that implements authenticated channels in the UC environment is used to bridge this difference.

Communication model: There are a few differences in the rules regarding the communication between machines in both models. We list them here.

In the SUC model, the adversary controls the flow of messages between machines using the router. The messages between protocol parties and also between parties and ideal functionalities are transferred via the router. For each incoming message, the router delivers the message only after being instructed to do so by the adversary. The only exception is messages between the environment and the adversary; these are transferred directly. Message delivery is carried out by the router in the following manner. First the router sends the message public header to the adversary, and keeps the original message inside a queue. If the adversary sends a header to the router, the router checks if a message with the same header exists in the outgoing queue, and if it finds such message, it sends the message to the designated recipient.

In contrast, in the UC model, the adversary controls the flow of messages directly and there is no router. The parties are connected in a star configuration via the adversary, and message delivery is carried out directly by the adversary. However, the adversary only controls the flow of messages between *protocol parties.* Ideal functionalities are invoked as subroutines by the protocol parties, and messages between parties and ideal functionalities are transferred directly via the input tape, see Figure 3. Thus, the adversary *cannot* influence the scheduling of these messages.

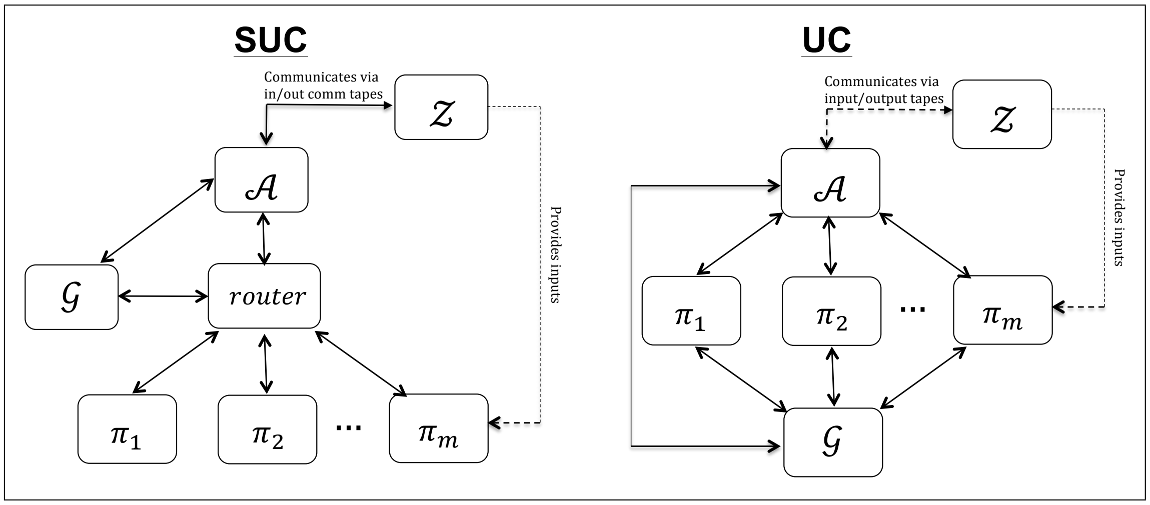


Figure 3: Differences in the communication models

In the SUC model the adversary and environment communicate via the *incoming/outgoing communication* tapes. In contrast, in the UC model the adversary and the environment communicate via their *input/output* tapes. Specifically, the environment provides input for the adversary and reads its output.[7]

Observe that since we emulate the authentication of messages in the UC model using the $\mathcal{F}_{\text{AUTH}}$ ideal functionality, the messages between protocol parties are not transferred via the adversary. However, the $\mathcal{F}_{\text{AUTH}}$ functionality itself asks permission from the adversary to process inputs and return outputs to the parties. Thus, the effect is the same as in the SUC model. This enables us to overcome the main difference regarding the communication flow.

3.2 General Transformation of ITMs

We will first show how to transform Interactive Turing Machines from the SUC model to ITMs as described in the definition of the full UC model.

3.2.1 Run-Time Transformation

Before we present the basic transformation of machines from the SUC model to the UC model, we provide the idea behind its design. First, the transformation works on the code tape of machine M and not on the ITM M itself. That is, a machine in the SUC model has different tapes from a machine in the UC model. Instead of copying data between different tapes (let's say, copy the content of the security tape of the original machine to the beginning of the input tape of the converted machine) we simply initialize a new machine $M' \in ITM^{UC}$ with a modified version of the code of M. This works because in the full UC model ITMs are invoked dynamically, and the way to invoke a new machine with code C_M and input x is by issuing an external write instruction $\textsf{external-write}(\mu, x)$ where $\mu := (C_M, id)$ is the identity string corresponding to the new machine.

Second, M' applies an input check in order to ensure a polynomial bound on its runtime. The original machine is polynomial in the security parameter n and inputs $x_1, \ldots, x_\ell$. The problem is that the runtime complexity of a UC machine is measured differently as mentioned in section (3.1), and must be polynomial in $\tilde{n} = n_I - n_O$. We generally cannot control n_O since it depends on the code of machine M. We therefore increase n_I artificially by providing longer input to each machine. We then encapsulate the original code (known to be bounded in $q(\sum_{i=1}^{\ell} |x_j| + n)$ steps) in a separate section of the converted code called **the core**. We also add a new section to the code called **the shell**. The shell preprocesses the inputs $x'_1, \ldots, x'_\ell$ of the converted machine, derives new (effective) inputs $x_1, \ldots, x_\ell$, and passes them to the core for actual processing.

Definition 3.1 *Let $C_M \in \{0,1\}^*$ be the code tape of a universal Interactive Turing Machine $M \in PPT^{SUC}$, and let $q(\cdot)$ be the polynomial bounding the run time of M (i.e. on inputs $x_1, \ldots, x_\ell$ and security parameter n, M performs at most $q(\sum_{i=1}^{\ell} |x_i| + n)$ steps). We define the* runtime transformation *$T_R : \{0,1\}^* \times \mathbb{Z}[x] \to \{0,1\}^*$ in the following manner. $T_R(\cdot)$ receives for input a code tape C_M of machine M in the SUC model and a polynomial $q \in \mathbb{Z}[x]$, and generates the code $C_{M'}$ of machine M' in the full UC model, as follows:*

- *Upon receiving the i^{th} input from the environment:*

[7] Obviously, since the ability to provide input to another machine is limited in the UC framework via the polynomial time bound, this mechanism inherently limits the communication between the environment and the adversary; Such a limit does not exist in SUC. However, since we prove that protocols secure in SUC are also secure in UC, this has little significance.

1. $T_R(C_M)$ checks that $x'_i = (1^{k_i}, 0, x_i)$ for some $k_i \in \mathbb{N}$.

2. $T_R(C_M)$ checks that $\sum_{j=1}^{i} k_j = q\left(\sum_{j=1}^{i} |x_j| + n\right)$.[8]

3. If yes to both, then $T_R(C_M)$ saves input x_i for future processing and immediately activates the environment (with an empty external write*). Otherwise, $T_R(C_M)$ halts.*

- *On its next activation $T_R(C_M)$ runs the code C_M on input x_i.*

The machine M' that runs the code $T_R(C_M)$ is called the correlated UC machine of M*, and we denote $T_R(M)$ by M'.*

Explanation of the transformation:

- For each input x'_i, M' performs the following checks:

  1. First, M' checks that the input begins with an unary string of arbitrary size k_i (called the token extension of the i^{th} input), followed by a zero, followed by another arbitrary string x_i (called the i^{th} effective input). The token extension part is needed for the input check (the shell) while the effective input part is transferred as is to the original code (the core).
  2. M' checks that the total number of token extension bits received until now is enough to guarantee that it runs in time that is polynomial in $\tilde{n}$. The token extensions k_i are used to increase n_I without increasing n_O as well.

- If these checks passes, x_i constitutes a valid input and should be processed. However, this input is only processed on the next activation. This is because in SUC, when the environment $\mathcal{Z}$ writes input to a protocol party, this does not complete its activation. In addition, the adversary is always activated immediately after the environment in the SUC framework. In contrast, in the UC framework each input constitutes an external write and by definition causes the activation of $\mathcal{Z}'$ to end, and the party receiving the input is then activated. In order to to simulate the SUC behaviour in UC we force the UC protocol parties to return the control to the environment before processing the input.

Our first result shows that the basic code transformation $T_R(\cdot)$ preserves the runtime complexity of polynomial-time machines.

Lemma 3.2 *Let $M \in PPT^{SUC}$ be a polynomial-time ITM in the SUC model, and let C_M be the code tape of M. Then machine $M' \in ITM^{UC}$ that runs the code $T_R(C_M)$ is locally p-bounded for some polynomial $p(\cdot)$ in the full UC model.*

Proof: In order to prove that M' is locally p-bounded for some polynomial $p(\cdot)$, we need to show that at any point during an execution of M', the overall number of computational steps is at most $p(\tilde{n})$ where $\tilde{n} = n_I - n_O$, n_I is the overall number of bits written so far on M' input tape, and n_O is the number of bits written by M so far to input tapes of ITM instances.

We begin by determining the size of $\tilde{n}$. We know that M runs in polynomial time and is bounded by $q(\cdot)$. Assuming the security parameter is n and that M already received ℓ reactive inputs

[8]It is possible to relax this condition so that $\sum_{j=1}^{i} k_j \geq q(\sum_{i=1}^{i} |x_j| + n)$ as long as $\sum_{j=1}^{i} k_j = O(q(\sum_{i=1}^{i} |x_j| + n))$. Nevertheless, the transformation also works with strict equality.

$x_1, \ldots, x_\ell$, the amount of steps made by M is at most $q(\sum_{j=1}^{\ell} |x_j| + n)$. The core of M' runs the same number of steps as M, and so the maximum number of bits passed as input to other machines is also $q(\sum_{j=1}^{\ell} |x_j| + n)$. The shell itself does not write on other machines' input tapes, and so does not lose tokens. Hence, $n_O \leq q(\sum_{j=1}^{\ell} |x_j| + n)$. The input of M' is of size $n_I = n + \sum_{j=1}^{\ell} |x_j| + \sum_{j=1}^{\ell} k_j$. Now, in every invocation the shell checks that $\sum_{j=1}^{\ell} k_j = q\left(\sum_{j=1}^{\ell} |x_j| + n\right)$, which is equivalent to verifying that $\sum_{j=1}^{\ell} k_j - n_O \geq 0$. Hence, we can conclude that for ℓ reactive inputs:

$$\tilde{n} = n_I - n_O = n + \sum_{j=1}^{\ell} |x_j| + \sum_{j=1}^{\ell} k_j - n_O \geq n + \sum_{j=1}^{\ell} |x_j|$$

where the inequality follows from the assumption that $\sum_{j=1}^{\ell} k_j - n_O \geq 0$ (since otherwise the machine halts immediately).

It remains to show that M' runs in time $p(\tilde{n})$ for some polynomial p. For each new reactive input, the shell performs an input check on the input tape; this check evaluates the polynomial $q(\cdot)$ on the effective inputs, and is therefore quadratic in the size of the input tape n_I. Regarding the runtime of the core, M' simply runs M (for simplicity we assume that this simulation of M by M' is quadratic in the runtime of M). We denote by $\mathsf{steps}(M')$ the total number of steps made by M' and by $\mathsf{steps}_i(M')$ the steps made in the ith invocation (when receiving the ith input). Since $\ell \leq \Sigma_{j=1}^{\ell} |x_j| + n \leq \tilde{n}$ we have:

$$\begin{aligned}
\mathsf{steps}(M') &= \sum_{i=1}^{\ell} \left(\mathsf{steps}_i(\mathrm{shell}(M')) + \mathsf{steps}_i(\mathrm{core}(M'))\right) \\
&= \sum_{i=1}^{\ell} \left(\left(n + \sum_{j=1}^{i} |x_j| + q\left(n + \Sigma_{j=1}^{i} |x_j|\right) \right)^2 + q^2\left(n + \Sigma_{j=1}^{i} |x_j|\right) \right) \\
&\leq \ell \cdot \left(\left(n + \sum_{j=1}^{\ell} |x_j| + q(n + \Sigma_{j=1}^{\ell} |x_j|) \right)^2 + q^2\left(n + \Sigma_{j=1}^{\ell} |x_j|\right) \right) \\
&\leq \ell \cdot \left((\tilde{n} + q(\tilde{n}))^2 + q^2(\tilde{n}) \right) \\
&\leq \tilde{n} \cdot \left((\tilde{n} + q(\tilde{n}))^2 + q^2(\tilde{n}) \right) \\
&= \mathrm{poly}(\tilde{n}).
\end{aligned}$$

We remark that in the case that for some input x_j the input check did not pass, the machine is also polynomial in $\tilde{n}$. This holds because the fact that the jth input was checked implies that the input check was successful for the first $j-1$ reactive inputs. Thus, until receiving x_j machine M' ran in $\mathrm{poly}(\tilde{n})$ time. Next, checking x_j takes $(n_I)^2 = \mathrm{poly}(\tilde{n})$ steps and the shell halts M' immediately after (without writing any output bits). Thus, M' runs in $poly(\tilde{n})$ in this case as well. We conclude that M' is locally p-bounded for some polynomial p. ∎

3.2.2 Communication and Protocol Party Transformation

As we described previously, in order to transform a protocol party from the SUC model to the full UC model, it is not enough to ensure polynomial runtime complexity since there are also differences between the models in the way protocol parties communicate between each other. Since the protocol parties in the SUC model communicate using authenticated channels, we need the correlated machines of protocol parties to use the $\mathcal{F}_{\text{AUTH}}$ ideal functionality. This transformation modifies a protocol party to use $\mathcal{F}_{\text{AUTH}}$ in order to emulate authenticated channels.

Definition 3.3 *We define the* communication transformation $T_C : \{0,1\}^* \to \{0,1\}^*$ *in the following manner.* $T_C(\cdot)$ *receives for input a code tape* $C_{P'}$ *of machine* $P' \in ITM^{UC}$ *and generates the code tape* $C_{P''}$ *of machine* $P'' \in ITM^{UC}$*.* $T_C(C_{P'})$ *runs* $C_{P'}$ *internally. Then,*

- *When* $C_{P'}$ *wishes to write an outgoing message* m *with addressing information* $\sigma = (S, R)$ *on its outgoing communication tape and activate the router,* $T_C(C_{P'})$ *routes the message in the following way:*
  - *If the destination* R *is a protocol party,* $T_C(C_{P'})$ *performs an external write on the input tape of functionality* $\mathcal{F}_{\text{AUTH}}$ *with input* (Send, sid, m) *where* $sid = (P', R, sid')$ *and* sid' *is the identifier of the current UC execution.*
  - *If the destination* R *is an instance of ideal functionality* $\mathcal{G}$*, machine* $T_C(C_{P'})$ *performs an external write of* m *directly on the input tape of the corresponding instance of* $\mathcal{G}'$ *in the UC execution.*
- *When* $C_{P'}$ *wishes to read an incoming message from a protocol party on its incoming communication tape,* $T_C(C_{P'})$ *reads an output from its subroutine output tape from* $\mathcal{F}_{\text{AUTH}}$*. When* $C_{P'}$ *wishes to read a message from* $\mathcal{G}'$*, machine* $T_C(C_{P'})$ *reads an output from its subroutine output tape from* $\mathcal{G}'$*.*
- $T_C(C_{P'})$ *ignores messages on its incoming communication tape. In essence, this means that* P'' *does not accept any unauthenticated messages.*

Finally, we define the protocol party transformation $T_P : \{0,1\}^* \times \mathbb{Z}[x] \to \{0,1\}^*$ *as* $T_P := T_C \circ T_R$*, where* $T_R(\cdot)$ *is the runtime transformation.*

We stress that applying transformation $T_C(\cdot)$ to an arbitrary locally p-bounded machine in the UC model does *not* necessarily result in a locally p-bounded machine. This is due to the fact that message are rerouted from outgoing communication tapes to input tapes, thus increasing n_O. Nevertheless, as we show in the following lemma, when T_C is applied specifically to a machine P' that is derived by applying the runtime transformation T_R to an SUC machine P, then UC polynomial-time is preserved. This is due to the fact that T_R ensures that P' has enough tokens, even if all P does is write to its output tapes.

Lemma 3.4 *Let* $M \in PPT^{SUC}$ *be a polynomial-time ITM in the SUC model, and let* C_M *be the code tape of* M*. Then machine* $M'' \in ITM^{UC}$ *that runs the code* $T_P(C_M) = T_C(T_R(C_M))$ *is locally* p*-bounded for some polynomial* p*.*

Proof: From the previous lemma we know that machine M' that runs the code $T_R(C_M)$ is locally p-bounded. We need to prove that M'' that runs the code of $T_C(C_{M'})$ is also locally p-bounded. There are 3 ways in which the code of M'' differs from the code of M':

1. When M' writes a message m on its outgoing communication tape, M'' writes it either on the input tape of $\mathcal{F}_{\text{AUTH}}$ or another ideal functionality $\mathcal{G}'$. The overhead of changing the instruction is constant and thus $\mathsf{steps}(M'') = O(\mathsf{steps}(M'))$. However, the change of tapes increases n_O which in turn makes $\tilde{n} = n_I - n_O$ smaller. However, since $T_C(\cdot)$ is applied to $M' = T_R(M)$, the input check made by the shell of M'' ensured that each reactive input received by M'' is padded with enough tokens even when M'' writes the maximum input possible to other machines. Thus, even for arbitrary number of outgoing messages there are enough tokens provided to M''.

2. When M' reads an incoming message from the incoming communication tape, M'' reads an incoming message from the subroutine output tape, and continues the same way. In this case $\mathsf{steps}(M'') = \mathsf{steps}(M')$.

3. M'' ignores messages on its incoming communication tape, but that only subtracts computational steps. Therefore in this case $\mathsf{steps}(M'') \leq \mathsf{steps}(M')$.

This completes the proof. ■

3.2.3 Transformation of Ideal Functionalities

We will now show how to transform an ideal functionality in the SUC model to an ideal functionality in the full UC model. As with protocol parties, we apply a set of macros to the code of $\mathcal{F}$, so that we can simulate the SUC adversary's power to control the flow of messages to the ideal functionality via the router.

Observe that SUC protocol parties communicate with an ideal functionality via the router and hence via communication tapes. When we transform an ideal functionality to the UC-environment, it is executed as a subroutine by the protocol parties, and thus receives the communication as input via the input tape. For this reason, in the following definition the converted functionality treats inputs as incoming messages and outputs as outgoing messages. Recall that converted protocol parties send messages directly to the ideal functionality in UC, and not via the router as in SUC.

Definition 3.5 *Let $\mathcal{F}$ be an ideal functionality in the SUC model. We define $\mathcal{F}' = \phi(\mathcal{F})$ to be the ideal functionality that works as follows:*

1. *Upon receiving an incoming message from party P via the input tape (input, sid, x) where $sid = (pid_P, sid_G)$:*
   - *Send the adversary $(\mathsf{input}, sid, H_{\mathcal{F}}(x))$, where $H_{\mathcal{F}}(x)$ is the public header of the message containing x.*
   - *Place (input, sid, x) in the delayed inputs queue.*

2. *Upon receiving a message (input, sid, m) from the adversary, if there is a message (input, sid, x) in the inputs queue (i.e., with the same sid) and $m = H_{\mathcal{F}}(x)$, then:*
   - *If $sid = (pid_P, sid_G)$ then set the origin of the input to party P.*
   - *Process the input x from party P using the original code of $\mathcal{F}$.*

   Else ignore the message.

3. *Upon generating a pending output* (sid_G, y) *to party* P*:*
   - *Set* $sid = (pid_P, sid_G)$*.*
   - *Send to the adversary* $(\mathsf{output}, sid, H_{\mathcal{F}}(y))$*, where* $H_{\mathcal{F}}(y)$ *is the public header of the message containing* y*.*
   - *Place the output* $(\mathsf{output}, sid, y)$ *in the delayed outputs queue.*
4. *Upon receiving a message* $(\mathsf{output}, sid, m)$ *from the adversary, if there is a message* $(\mathsf{output}, sid, y)$ *in the inputs queue (i.e., with the same sid) and* $m = H(y)$*, then:*
   - *If* $sid = (pid_P, sid_G)$ *then set the destination of the output to party* P*.*
   - *Write the output* y *on the subroutine output tape of the party* P*.*

   Else ignore the message.
5. *Upon receiving a message* $(\mathsf{corrupt}, P)$ *from the adversary,* $\mathcal{F}'$ *follows the* standard corruption *procedure specified in [7]. That is, it notifies* P *that it is corrupted and sends all inputs and outputs of* P *to the adversary.*
6. *Upon receiving a message* $(\mathsf{CorruptOutput}, sid, m)$ *from the adversary, if there is a message* $(\mathsf{output}, sid, y)$ *in the outputs queue (with the same sid) and* $sid = (pid_P, sid_G)$ *and* P *was already corrupted, then write the output* m *on the subroutine output tape of the party* P*. Else ignore the message.*
7. *Upon receiving a message* $(\mathsf{message}, sid, m)$ *from the adversary, pass the message* m *to* $\mathcal{F}$ *without any change (this message is part of the arbitrary communication allowed between the adversary and an ideal functionality).*
8. *Upon generating an outgoing message* m *to the adversary, pass the message* $(\mathsf{message}, sid, m)$ *to the adversary (this message is part of the arbitrary communication allowed between the adversary and an ideal functionality).*

We call ϕ *the* functionality transformation *and* $F' = \phi(F)$ *is called the* converted functionality*. The code of* $\mathcal{F}$ *executed by* $\mathcal{F}'$ *is called the* core of $\mathcal{F}'$*.*

As above, we begin by claiming that $\mathcal{F}$ is polynomial-time in the SUC model, then $\phi(\mathcal{F})$ is polynomial-time in the UC model.

Lemma 3.6 *Let* $\mathcal{F} \in PPT^{SUC}$ *be an ideal functionality machine in the SUC model with runtime that is bounded by polynomial* q*. Then machine* $\mathcal{F}' \in ITM^{UC}$ *that runs the code* $\phi(\mathcal{F})$ *is locally* q*-bounded.*

Proof: By assumption, $\mathcal{F}$ is PPT in SUC. Therefore, for any series of ℓ reactive inputs $x_1, \ldots, x_\ell$ and security parameter n, $\mathcal{F}$ performs at most $q(\sum_{i=1}^{\ell} |x_j| + n)$ steps. The only difference between $\mathcal{F}$ and $\mathcal{F}'$ is that $\mathcal{F}'$ requests approval from the adversary to process each input and deliver each output, thus adding only a constant factor to the processing of each message. Furthermore, since $\mathcal{F}$ is a machine in the SUC model it cannot invoke other machines, and thus $\mathcal{F}'$ does not invoke other machines (observe that the output written by $\mathcal{F}'$ is to the subroutine output tape of a party and thus does not contribute to n_O). We conclude that $n_O = 0$ and so $\tilde{n} = n_I$. Since the input tape of $\mathcal{F}'$ is of size $n_I = \sum_{i=1}^{\ell} |x_j| + n$, we conclude that $\mathsf{steps}(\mathcal{F}') \leq O(q(\sum_{i=1}^{\ell} |x_j| + n)) = O(q(\tilde{n}))$. Therefore $\mathcal{F}'$ is q-bounded. ■

3.2.4 Preserving the PPT Property of Protocols

So far we have shown that each machine in isolation is locally p-bounded. We now extend the treatment to protocol executions that involve multiple machines, as well as an environment and an adversary. In the full UC definition, if a machine M is locally p-bounded for some polynomial $p(\cdot)$, and in addition each external write request carried out by M specifies a recipient ITM which is locally p-bounded, then it is PPT. Furthermore, a protocol is PPT if it is PPT as an ITM, or less formally, if every ITM that is part of the protocol main instance (following the protocol specification) is PPT. However, we have already shown that all transformed protocol parties are locally p-bounded and that any transformed ideal functionality is locally p-bounded. The only additional entity is $\mathcal{F}_{\text{AUTH}}$ which is trivially locally p-bounded. We have the following corollary:

Corollary 3.7 *Let $\pi_1, \ldots, \pi_m$ be a set of machines defined by the specification of a PPT protocol π running in the SUC execution environment. If the adversary and environment machines $\mathcal{A}'$ and $\mathcal{Z}'$, respectively, are PPT in the full UC framework, then machines $T_P(\pi_1), \ldots, T_P(\pi_m)$ are parties of a PPT protocol in the full UC model.*

3.3 SUC security implies UC security

We now proceed to prove that any protocol that is secure in the SUC model is also secure in the full UC model. We first present the proof outline, then define the transformations that are needed for the proof, and finally we present the proof itself.

3.3.1 Proof Outline

In order to prove that SUC security implies UC security, we work in the following way. We first obtain a UC protocol π' from the SUC protocol π; the UC protocol π' is obtained by transforming the protocol parties in π with a transformation T_P of SUC protocol parties to UC protocol parties. Likewise, we define a transformation ϕ from SUC ideal functionalities $\mathcal{F}$ to UC ideal functionalities $\mathcal{F}'$. We then show that π' securely realizes $\mathcal{F}'$ in the full UC framework. This is proven by contradiction, by showing that if the converted protocol $\pi' = T_P(\pi)$ does not UC-realize the converted functionality $\mathcal{F}' = \phi(\mathcal{F})$, then π does not SUC-realize the functionality $\mathcal{F}$.

In order to do this, we begin with a UC adversary $\mathcal{A}'$ for π'. We then transform $\mathcal{A}'$ attacking π' into a corresponding SUC adversary $\mathcal{A}$ attacking π. By the assumption that π is SUC-secure, we have that there exists a simulator $\mathcal{S}$ for $\mathcal{A}$ in the SUC ideal world with $\mathcal{F}$. We then apply another transformation to transform $\mathcal{S}$ into a simulator for $\mathcal{A}'$ in UC ideal world with $\mathcal{F}'$. Finally, we show that $\mathcal{S}'$ is a "good simulator" for every UC environment $\mathcal{Z}'$. We do this by showing that if not, then we can construct an SUC environment $\mathcal{Z}$ (that internally simulates the UC world for $\mathcal{Z}'$) that contradicts the assumption that $\mathcal{S}$ is a "good simulator" for $\mathcal{A}$ in the SUC framework; this construction of $\mathcal{Z}'$ from $\mathcal{Z}$ is via yet another transformation. This proof strategy is depicted in Figure 4.

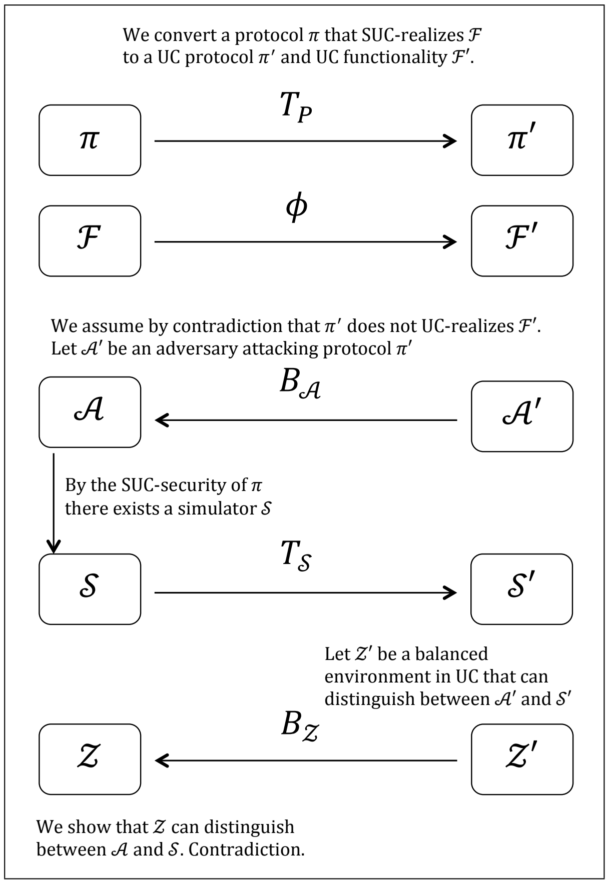

We convert a protocol π that SUC-realizes $\mathcal{F}$ to a UC protocol π' and UC functionality $\mathcal{F}'$.
π
T_P
π'
$\mathcal{F}$
ϕ
$\mathcal{F}'$
We assume by contradiction that π' does not UC-realizes $\mathcal{F}'$. Let $\mathcal{A}'$ be an adversary attacking protocol π'
$\mathcal{A}$
$B_{\mathcal{A}}$
$\mathcal{A}'$
By the SUC-security of π there exists a simulator $\mathcal{S}$
$\mathcal{S}$
$T_{\mathcal{S}}$
$\mathcal{S}'$
Let $\mathcal{Z}'$ be a balanced environment in UC that can distinguish between $\mathcal{A}'$ and $\mathcal{S}'$
$\mathcal{Z}$
$B_{\mathcal{Z}}$
$\mathcal{Z}'$
We show that $\mathcal{Z}$ can distinguish between $\mathcal{A}$ and $\mathcal{S}$. Contradiction.

Figure 4: Transformations required for the proof of security

3.3.2 Additional Transformations for the Proof of Security

We have already defined the protocol party and ideal functionality transformations. For the proof outlined above, we still need to define the following transformations:

1. $B_{\mathcal{A}} : \mathcal{A}' \to \mathcal{A}$ – this transformation transforms a UC adversary into an SUC adversary. Since the UC adversary is more powerful, the transformation internally emulates an entire UC runtime and uses an internal instance of $\mathcal{A}'$.

2. $T_{\mathcal{S}} : \mathcal{S} \to \mathcal{S}'$ – this transformation transforms an SUC simulator into an UC simulator by replacing the communication of the SUC ideal adversary with the router with communication of the adversary directly with the ideal functionality, thus changing the control of message flow to suit the UC environment.

3. $B_{\mathcal{Z}} : \mathcal{Z}' \to \mathcal{Z}$ – this transformation transforms a UC environment into an SUC environment. Similarly to $B_{\mathcal{A}}$, the transformation internally emulates an entire UC runtime and uses an internal instance of $\mathcal{Z}'$.

The ideal simulator transformation $T_{\mathcal{S}}$ (SUC to UC): We now present a transformation from an SUC ideal simulator to a UC ideal simulator. The only difference is with respect to the control over the delivery of messages between the other machines. Specifically, instead of interacting with the router in order to control the delivery of messages, the transformed adversary interacts with a transformed ideal functionality $\mathcal{F}' = \phi(\mathcal{F})$. Observe than $\mathcal{F}'$ includes message scheduling capabilities that are equivalent to an SUC router machine.

Definition 3.8 *We define the adversary transformation $T_{\mathcal{S}} : \{0,1\}^* \to \{0,1\}^*$, in the following manner. $T_{\mathcal{S}}(\cdot)$ is given a code tape $C_{\mathcal{S}}$ of adversarial machine $\mathcal{S}$ in the SUC model, and generates the code $C_{\mathcal{S}'}$ of machine $\mathcal{S}'$ in the full UC model, as follows:*

- *Upon receiving an incoming public header (a message with no private content) on the incoming communication tape, $T_{\mathcal{S}}(C_{\mathcal{S}})$ processes the header with the function in $C_{\mathcal{S}}$ that processes incoming public headers from the router.*
- *When $C_{\mathcal{S}}$ wishes to instruct the router to deliver a message m with header $H_{\mathcal{F}}(m)$, $T_{\mathcal{S}}(C_{\mathcal{S}})$ writes the message $(\mathsf{output}, sid, H_{\mathcal{F}}(m))$ on the incoming communication tape of the functionality $\mathcal{F}'$ instead.*
- *When $C_{\mathcal{S}}$ wishes to corrupt a protocol party, $T_{\mathcal{S}}(C_{\mathcal{S}})$ passes the corruption message to the ideal functionality $\mathcal{F}'$ directly. We note that the message from $C_{\mathcal{S}}$ to the environment in the SUC model is not passed by $T_{\mathcal{S}}(C_{\mathcal{S}})$ to the environment in the UC model (a different mechanism is used for this purpose in the UC model).*
- *When $C_{\mathcal{S}}$ wishes to write a message for the environment on the outgoing communication tape, $T_{\mathcal{S}}(C_{\mathcal{S}})$ writes the message on the output tape instead.*

We prove that $T_{\mathcal{S}}(\mathcal{S})$ is polynomial-time in the UC model.

Lemma 3.9 *Let $\mathcal{S} \in PPT^{SUC}$ be the adversary/simulator in a* SUC-IDEAL *execution. Then, machine $T_{\mathcal{S}}(\mathcal{S})$ is PPT in the* IDEAL *execution of the UC model.*

Proof: First, observe that since $\mathcal{S}$ does not write any messages on the input tapes of other parties (since this is not possible in the SUC model), it follows that $\mathcal{S}'$ also does not write on any other machine's input tape. It therefore follows that $n_O = 0$, and so for every possible execution of $\mathcal{S}'$ with security parameter n and ℓ reactive inputs we have $\tilde{n} = n_I = \sum_{j=1}^{\ell} |x_j| + n$. Next, since $\mathcal{S}$ is PPT in the SUC model, its runtime is bounded by some polynomial $q(\cdot)$. This implies that $\mathcal{S}$ performs at most $q(\sum_{j=1}^{\ell} |x_j| + n)$ steps. The transformation $T_{\mathcal{S}}(\cdot)$ adds only a constant factor to the processing of each message and so $\mathsf{steps}(\mathcal{S}') \leq O(q(\sum_{j=1}^{\ell} |x_j| + n)) = O(q(\tilde{n}))$. Thus, $\mathcal{S}'$ is locally q-bounded, and since it never invokes other machines it is also PPT. ■

The UC to SUC transformations – preliminaries: As we have mentioned, we need to show how to transform a UC adversary and a UC environment to the SUC framework. These transformation are essentially black-box emulations since we need to emulate the entire UC execution environment inside the SUC machine generated by these transformations. The emulation is required in order to to properly handle the dynamic generation of UC machines. For example, there may be adversaries that run subprotocol adversary machines as part of their execution.

In order to understand the design decisions behind these transformations we present a recap of the external write mechanism in the UC framework. Each such instruction is formatted as external-write(S, R, t, m) where $S = (id_S, C_S)$ is the machine executing the instruction, $R = (id_R, C_R)$ is the recipient machine, t is the name of the tape of R machine S wishes to write on, and m is the message to write. The following outcomes are possible for $\textsf{external-write}(S, R, t, m)$:

- *The control function did not allow the instruction:* in this case the instruction is ignored.
- *There is no machine in the execution with id* $= id_R$*:* in this case R is dynamically invoked and the message is sent.
- *There is already a machine* R' *in the execution with* $id_{R'} = id_R$*:* in this case,
  - If the the code of R' is not the same as the code of R and t is considered as a trusted tape (see [4, 7]), S enters a special error state.
  - Otherwise the message is sent to R'.

The adversary black-box transformation $B_{\mathcal{A}}$ (UC to SUC): In this transformation we must address the cases of dynamic machine generation and special error states. Since the set of protocol parties is known to $\mathcal{A}$ up front, $\mathcal{A}$ can simulate the proper response from the control function when the code of $\mathcal{A}'$ refers an external machine.

Definition 3.10 *We define the adversary reverse transformation $B_{\mathcal{A}} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, in the following manner. $B_{\mathcal{A}}(\cdot)$ is given a code tape $C_{\mathcal{A}'}$ of a UC adversary machine $\mathcal{A}'$ and the code tape $C_{control}$ of a UC control function machine, and generates the code $C_{\mathcal{A}}$ of machine $\mathcal{A} \in ITM^{SUC}$. $\mathcal{A}$ internally invokes an instance of $\mathcal{A}'$ and then works as follows:*

- *Upon receiving a message from the environment $\mathcal{Z}$ on its incoming communication tape, $\mathcal{A}$ writes the message on $\mathcal{A}'$'s input tape.*
- *When $\mathcal{A}'$ writes a value to its output tape, $\mathcal{A}$ sends it to the environment on its outgoing communication tape.*
- *When $\mathcal{A}$ receives an incoming message from the router or from an ideal functionality, $\mathcal{A}$ writes the message on the simulated incoming communication tape of $\mathcal{A}'$.*
- *When $\mathcal{A}'$ performs an external write to a machine $\mu = (id, \hat{M})$:*
  - *If $C_{control}$ disallows the message, the instruction is ignored.*
  - *If id is the designated identifier of an existing machine M and $C_{\hat{M}} = C_M$ then $\mathcal{A}$ treats it as an outgoing message (see below).*
  - *If id is the designated identifier of an existing machine M and $C_{\hat{M}} \neq C_M$, then $\mathcal{A}$ simulates $\mathcal{A}'$ receiving a special error message from the control function.*
  - *If id does not refer to an existing machine, $\mathcal{A}$ internally simulates for $\mathcal{A}'$ an invocation of the machine μ.*[9]
- *When $\mathcal{A}'$ writes an outgoing message to an already existing machine (using external-write):*

[9]In the UC model this generates a new subroutine machine for $\mathcal{A}'$. Since in SUC the protocol parties and ideal functionalities are fixed and known to all, this must be a subprotocol of the adversary.

  - *If the message is a corruption message of the form* $(\mathsf{corrupt}, id, p)$*, where* p *is the corruption parameter,* $\mathcal{A}$ *sends the message* $(\mathsf{corrupt}, id)$ *to party* id *via the router and also sends it to the environment.*
  - *If the message is written on the simulated incoming communication tape of a protocol party,* $\mathcal{A}$ *ignores the message (in the UC model* $\mathcal{A}$ *can write directly to a party, but this is not true in the SUC model; thus such messages are just ignored).*
  - *If the message is written on the simulated incoming communication tape of the functionality* $\mathcal{F}_{\text{AUTH}}$*,* $\mathcal{A}$ *sends the message to the router instead.*[10]
  - *If the message is written on the simulated incoming communication tape of an ideal functionality* $\mathcal{G}'$ *(where* $\mathcal{G}' \neq \mathcal{F}_{\text{AUTH}}$*),* $\mathcal{A}$ *sends the message to the corresponding ideal functionality* $\mathcal{G}$ *in the SUC execution.*[11]

The generated machine $\mathcal{A}$ *is called a* black-box shell *of adversary* $\mathcal{A}' \in PPT^{UC}$*.*

The environment black-box transformation $B_{\mathcal{Z}}$ (UC to SUC): The environment reverse transformation is simpler since it interacts only with the protocol parties and the adversary. It must however tokenize the inputs of the SUC protocol parties correctly or send the special HALT input.

Definition 3.11 *We define the environment reverse transformation* $B_{\mathcal{Z}} : \{0,1\}^* \times \{0,1\}^* \times (\mathbb{Z}[x])^m \to \{0,1\}^*$*, in the following manner.* $B_{\mathcal{Z}}(\cdot)$ *is given a code tape* $C_{\mathcal{Z}'}$ *of a UC environment machine* $\mathcal{Z}'$ *and the code tape* $C_{control}$ *of a UC control function machine and polynomials* $q_1, \ldots, q_m$*, and generates the code* $C_{\mathcal{Z}}$ *of machine* $\mathcal{Z} \in ITM^{SUC}$*.* $\mathcal{Z}$ *internally invokes an instance of* $\mathcal{Z}'$ *and then works as follows:*

- *Upon receiving a message on its incoming message tape from the adversary,* $\mathcal{Z}$ *writes the message on the simulated adversary's output tape to be read by* $\mathcal{Z}'$*.*
- *When* $\mathcal{Z}'$ *writes an input on the simulated input tape of the adversary,* $\mathcal{Z}$ *sends the value to the adversary on its communication tape.*
- *The output of* $\mathcal{Z}'$ *is set to be the output of* $\mathcal{Z}$*.*
- *When* $\mathcal{Z}'$ *writes the* i^{th} *(reactive) input* x'_i *on the simulated input tape of party* P_j*:*
  - $\mathcal{Z}$ *first checks that* x'_i *is formatted as* $x'_i = (1^{k_i}, 0, x_i)$*, where* x_i *is an arbitrary string, and that the requirement* $\sum_{j=1}^{i} k_j = q(\sum_{i=1}^{i} |x_j| + n)$ *holds, where* $k_1, \ldots, k_{i-1}$ *are the token extensions of previous reactive inputs passed to* P_j*, and* $x_1, \ldots, x_{i-1}$ *are the corresponding effective inputs.*
  - *If both checks passed successfully,* $\mathcal{Z}$ *passes the effective input* x_i *to party* P_j*.*
  - *Otherwise,* $\mathcal{Z}$ *passes* $HALT$ *to party* P_j*.*

The generated machine $\mathcal{Z}$ *is called a* black-box shell *of environment* $\mathcal{Z}' \in PPT^{UC}$*.*

[10]We assume that $\mathcal{F}_{\text{AUTH}}$ is never used in the SUC model. This is a reasonable assumption since it offers nothing beyond what the router already supplies.

[11]In the case $\mathcal{A}'$ operates in a $\mathcal{G}'$-Hybrid model where $\mathcal{G}' = \phi(\mathcal{G})$.

We show that these transformations preserve polynomial-time.

Lemma 3.12 *If $\mathcal{A}', \mathcal{Z}' \in PPT^{UC}$ then the corresponding black box shells $\mathcal{A}, \mathcal{Z}$ are PPT in the SUC environment.*

Proof: We prove the lemma for adversary $\mathcal{A}'$; the proof for environment $\mathcal{Z}'$ is identical. Since $\mathcal{A}' \in PPT^{UC}$, it follows that $\mathcal{A}'$ is p-bounded for some polynomial $p(\cdot)$. That is, for every $\ell \in \mathbb{N}$, and for every possible set of reactive inputs $x_1, \ldots, x_\ell$ and security parameter n, $\mathcal{A}'$ runs at most $p(n_I - n_O)$ steps, where $n_I = n + \sum_{j=1}^{\ell} |x_j|$ and $n_O \geq 0$. Since $p(n_I - n_O) \leq p(n_I)$, $\mathcal{A}'$ is also bounded in $p(n_I) = p(n + \sum_{j=1}^{\ell} |x_j|)$.

We now bound the runtime of $\mathcal{A}$. Let $x_1, \ldots, x_i$ be the series of inputs passed to $\mathcal{A}$ during its computation, and let n be the security parameter. Since $\mathcal{A}$ passes the inputs unchanged to the simulated copy of $\mathcal{A}'$, the content of the input tape of $\mathcal{A}'$ is (eventually) set to $n_I = n + \sum_{j=1}^{\ell} |x_j|$. Therefore, as we mentioned before, $\mathcal{A}'$ runs at most $p(n + \sum_{j=1}^{\ell} |x_j|)$ steps. Since $\mathcal{A}$ runs $\mathcal{A}'$ internally, and this simulation can be carried out in quadratic time, we have that $\mathcal{A}$ runs in time that is polynomial in $n + \sum_{j=1}^{\ell} |x_j|$, as required. ■

3.3.3 Proof of Security

We are now ready to prove that if a protocol is secure in the SUC $\mathcal{G}$-hybrid model, then its converted protocol is also secure in the full UC $(\phi(\mathcal{G}), \mathcal{F}_{\text{AUTH}})$-hybrid model. This means that is suffices to prove security for the SUC protocol and then UC security is automatically derived.

Theorem 3.13 *Let π be an SUC protocol running with up to m parties and let $\mathcal{F}$ be an ideal functionality. If π* SUC-securely computes *$\mathcal{F}$ in the $\mathcal{G}$-Hybrid model, then $T_P(\pi)$* UC-realizes *$\phi(F)$ in the $(\phi(\mathcal{G}), \mathcal{F}_{\text{AUTH}})$-hybrid model.*

Proof: Let $\pi' = T_P(\pi)$ be the converted protocol. Let $\mathcal{F}' = \phi(\mathcal{F})$ and $\mathcal{G}' = \phi(\mathcal{G})$ be the converted functionalities of $\mathcal{F}$ and $\mathcal{G}$. Our goal is to prove that π' UC-realizes the ideal protocol of $\mathcal{F}'$.

We assume by contradiction that π' *does not* UC-securely compute $\mathcal{F}'$. Consequently, there exists a PPT adversary $\mathcal{A}'$ such that for any PPT simulator $\mathcal{S}'$ there is a balanced PPT environment $\mathcal{Z}'$ that is able to distinguish between an execution with π' and $\mathcal{A}'$ and an execution with the ideal protocol of $\mathcal{F}'$ and $\mathcal{S}'$. For the sake of clarity we call $\mathcal{A}'$ a security contradictor adversary, because the existence of such adversary contradicts the security of protocol π'. We will show that if such an adversary $\mathcal{A}'$ exists, then the original protocol π does not SUC-securely compute $\mathcal{F}$.

Let $\mathcal{A}'$ be the security contradictor adversary of π' and let $\mathcal{A}$ be the black-box shell of $\mathcal{A}'$ in the SUC model. By the security assumption of π, there exists an ideal-model adversary $\mathcal{S} \in PPT^{SUC}$ such that for *every* balanced environment $\mathcal{Z} \in PPT^{SUC}$ there exists a negligible function $\mu(\cdot)$ so that for every $z \in \{0,1\}^*$ and every $n \in \mathbb{N}$,

$$\left| \Pr[\text{SUC-IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(n,z) = 1] - \Pr[\text{SUC-HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}(n,z) = 1] \right| \leq \mu(n). \qquad (1)$$

Let $\mathcal{S}' = T_{\mathcal{S}}(\mathcal{S})$ be the transformed UC machine of $\mathcal{S}$; $\mathcal{S}'$ is a PPT simulator in the UC environment. By the assumption that $\mathcal{A}'$ contradicts the security of π' there exists a distinguisher balanced environment $\mathcal{Z}' \in PPT^{UC}$ and $c, d \in \mathbb{N}$ such that for an infinite number of pairs (n, z) satisfying $n \in \mathbb{N}$ and $z \in \cup_{\kappa \leq n^d}\{0,1\}^\kappa$ we have:

$$\left|\Pr[\text{IDEAL}_{\mathcal{F}',\mathcal{S}',\mathcal{Z}'}(n,z) = 1] - \Pr[\text{HYBRID}^{\mathcal{G}',\mathcal{F}_{\text{AUTH}}}_{\pi',\mathcal{A}',\mathcal{Z}'}(n,z) = 1]\right| \geq n^c. \tag{2}$$

Let $\mathcal{Z}$ be the black-box shell of $\mathcal{Z}'$ in the SUC model. Since π SUC-securely computes $\mathcal{F}$, $\mathcal{Z}$ satisfies Eq. (1). We will use this to derive a contradiction.

We begin by proving that the associated hybrid distributions are identically distributed (Lemma 3.14), and then that the associated ideal distributions are identically distributed (Lemma 3.15).

Lemma 3.14 *Let $\pi, \pi', \mathcal{A}, \mathcal{A}', \mathcal{Z}, \mathcal{Z}'$ be as above. Then for every $d \in \mathbb{N}$:*

$$\left\{\text{SUC-HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}(n,z)\right\}_{n\in\mathbb{N}, z\in\cup_{\kappa\leq n^d}\{0,1\}^\kappa} \equiv \left\{\text{HYBRID}^{\mathcal{G}',\mathcal{F}_{\text{AUTH}}}_{\pi',\mathcal{A}',\mathcal{Z}'}(n,z)\right\}_{n\in\mathbb{N}, z\in\cup_{\kappa\leq n^d}\{0,1\}^\kappa}$$

Proof: In each execution we refer to the activations of the *environment* and *adversary* machines as adversarial activations; the activations of other machines are simply called activations. This distinction is used so we can prove that the adversarial machines have the same state and eventual outputs in both executions.

Each ensemble denotes the output of the respective environment ($\mathcal{Z}$ or $\mathcal{Z}'$) in an execution with parameters (n, z) where n is the security parameter and z is the auxiliary input of the environment. We will show that for any pair (n, z) the outputs of $\mathcal{Z}$ and $\mathcal{Z}'$ are the same. We will do so by induction on the adversarial activations in each execution.

Formally, we prove that for every j the internal state of *all machines* in the j^{th} adversarial activation is the same in both executions. More exactly, we show that the internal state of $\mathcal{Z}'$ (resp., $\mathcal{A}'$) in the full UC execution is identical to the internal state of the simulated $\mathcal{Z}'$ inside $\mathcal{Z}$ (resp., $\mathcal{A}'$ inside $\mathcal{A}$) in the SUC execution. In addition, the internal state of the core of each protocol party in the full UC execution is identical to the internal state of the corresponding protocol party in the SUC execution, and the internal state of the core of $\mathcal{G}'$ in the full UC execution is identical to the internal state of $\mathcal{G}$ in the SUC execution. Finally, the behavior of the router and $\mathcal{F}_{\text{AUTH}}$ is identical in both cases.

The first machine to be activated in both executions is the environment. Thus, the first activation is also the first adversarial activation. In the first activation of $\mathcal{Z}$, it runs $\mathcal{Z}'$ on input (n, z) and thus the internal state of $\mathcal{Z}'$ is the same in the UC and SUC executions. Since $\mathcal{Z}'$ is a UC environment, its first action is to write to the input tape of $\mathcal{A}'$. $\mathcal{Z}$ receives this message from its internal $\mathcal{Z}'$, and forwards it to the SUC adversary $\mathcal{A}$ without changes. $\mathcal{A}$ passes messages from $\mathcal{Z}$ to $\mathcal{A}'$ without any change. Thus, the internal states of $\mathcal{A}'$ in UC and $\mathcal{A}'$ inside $\mathcal{A}$ in SUC are the same at the end of the first activation. In their next activation, they will run on the same input exactly.

We assume that the internal state of *all machines* is identical at the end of the jth adversarial activation, and proceed to prove that this holds after the $(j+1)$th adversarial activation. We need to show that the series of activations between the jth and $(j+1)$th adversarial activations results in identical internal states. Since the internal states are the same after the jth adversarial activation, this implies that the same machine M is activated next[12]. We separate the discussion according to the machine that was activated in the j^{th} adversarial activation.

[12]otherwise the execution was ended at the end of the jth adversarial activation.

Case 1 – the environment is activated: $\mathcal{Z}/\mathcal{Z}'$ can provide input to a protocol party, or communicate with the their respective adversaries $\mathcal{A}/\mathcal{A}'$, or output their decision bit and end the execution.

- If M is the adversary $\mathcal{Z}$ and $\mathcal{Z}'$ passes the same message to their respective adversaries, and the next activation is also an adversarial activation with the same input provided to $\mathcal{A}$ and $\mathcal{A}'$. Thus this is a trivial case.
- If M is a protocol party:
  - If the correct number of tokens are provided by $\mathcal{Z}'$ then the same *effective input* is written in both cases. Since the parties of π' reactivates the environment immediately after receiving input (see definition 3.1) this does not end the activation, and $\mathcal{Z}/\mathcal{Z}'$ can provide input to other machines or to the adversary.
  - If not enough tokens are provided, $\mathcal{Z}$ does not pass this input and instead provides the special HALT input to the designated party; in UC, the protocol party's input check will fail. In both cases the party will halt and then reactivate the environment with an empty message. Thus, the next adversarial activation starts in the same state in both executions.
- If $\mathcal{Z}'$ writes to its output and halt, then $\mathcal{Z}$ writes the same bit and also halts.

Case 2 – the adversary is activated: By the inductive assumption, $\mathcal{A}'$ and $\mathcal{A}$ are activated on the same input. Given that they have the same internal state, $\mathcal{A}'$ inside $\mathcal{A}$ in SUC carries out the same operation as $\mathcal{A}'$ in UC. The following operations can be carried out by the adversary:

- *Corruption:* In UC, $\mathcal{A}'$ writes the corruption messages directly to the party π'_i that it wishes to corrupt. This machine then immediately notifies its parent (the machine that provides input to π'_i, in this case the environment) by writing the corruption message on its output tape. In the SUC execution $\mathcal{A}$ receives the corruption message internally from $\mathcal{A}'$, and sends it to the router and the environment. In both cases, the party is corrupted with the same corruption message and the environment is notified. Thus, the internal state of the machines after the series of activations spawned by a corruption is identical. In both the UC and SUC executions, the environment is activated next (see Section 2.4 for the exact sequence of steps of a corruption in the SUC model), completing this inductive step.
- *Routing:* If $\mathcal{A}'$ sends an outgoing message to the ideal functionality $\mathcal{F}_{\text{AUTH}}$, then $\mathcal{A}$ receives this outgoing message internally and sends its to the router instead. Both $\mathcal{F}_{\text{AUTH}}$ and the router follow the same rules regarding delivering the message, and provide exactly the same scheduling capabilities to their respective adversaries. Thus in both cases $\mathcal{A}$ and $\mathcal{A}'$ delivered the same message msg to their respective scheduler. The next machine to be activated after the router/$\mathcal{F}_{\text{AUTH}}$ might be:
  - *The adversary* will be activated immediately in case the instruction given to the router/$\mathcal{F}_{\text{AUTH}}$ is illegal (like deliver a message that is not in the buffer).
  - *Protocol party* π_i/π'_i will receive msg in case it was already in the buffer. π_i might generate output y_i thus activating the environment. Since π'_i generates the same output in both cases the environment is activated with the same internal state.

Otherwise π_i sends an outgoing message to another party/ideal functionality via the router. The adversary $\mathcal{A}$ is activated immediately after. π'_i sends the same outgoing message via $\mathcal{F}_{\text{AUTH}}$ and $\mathcal{A}'$.

  - *Ideal functionality $\mathcal{G}/\mathcal{G}'$* will receive *msg* in case it was already in the buffer. $\mathcal{G}$ can generate a delayed output (thus activating the router/$\mathcal{F}_{\text{AUTH}}$ and then the adversary) or can write an arbitrary message directly to the adversary (in which case the adversary is activated next). Since $\mathcal{G}'$ runs $\mathcal{G}$ internally they both generate the same message. Thus the adversaries are activated almost immediately with the same internal state.

- *Communication with the environment:* When $\mathcal{A}'$ sends a message to the environment, the exact same message is sent by $\mathcal{A}$ (albeit via a different tape), and the environments $\mathcal{Z}'$ and $\mathcal{Z}$ are activated next with the same message.
- *Arbitrary communication with the ideal functionality:* When $\mathcal{A}'$ sends a direct message to the ideal functionality $\mathcal{G}'$ (not a scheduling instruction), the exact same message is sent by $\mathcal{A}$ to the ideal functionality $\mathcal{G}$. We already showed that the adversaries are activated shortly after $\mathcal{G}/\mathcal{G}'$ are activated.
- *Direct Communication with protocol parties:* When $\mathcal{A}'$ attempts to send a message to a protocol party, $\mathcal{A}$ blocks the message since in the SUC model the adversary can only write via the router and the router will block any such message. It then activates $\mathcal{A}'$ again by sending it an empty message. Note also that in the transformation of the protocol party from the SUC to the UC model, we instructed π' to ignore any such messages from $\mathcal{A}'$ and to send back an empty message to $\mathcal{A}'$ so that it is activated again. Thus, the behavior in both executions is the same.
- *Subroutine generation:* If $\mathcal{A}'$ invokes a new machine with identity $\mu = (id, \hat{M})$ in the UC execution, then:
  - If *id* is a new identity, then $\mathcal{A}$ internally simulates the machine for $\mathcal{A}'$ and so the result is the same in both executions.
  - If *id* is the designated identifier of a protocol party which has *not* yet been generated and $\hat{M}$ is the protocol party's code, then $\mathcal{A}$ does nothing. In the UC execution this will generate a new machine but since a protocol party's input must come from the environment (and the convention in the UC model is that a protocol party verifies that its input was received from an external write from the environment), $\mathcal{A}'$ will not be able to do anything with this machine.
  - If *id* is the designated identifier of a protocol party which has *already* been generated and $\hat{M}$ is not the party's code, then $\mathcal{A}$ simulates $\mathcal{A}'$ receiving a special error message from the control function. This is the same as $\mathcal{A}'$ will receive in the UC execution after such an external write.
  - If *id* is the designated identifier of a protocol party which has *not* yet been generated and $\hat{M}$ is not the party's code, then in the UC execution, the environment $\mathcal{Z}'$ will receive a special error message when trying to generate the protocol party machine. This scenario is simulated identically by $\mathcal{A}$ and $\mathcal{Z}$, as specified in the transformation.

This completes the inductive step. We have shown that the internal state of all machines is the same in both executions (more exactly, the core of the UC-machine has the same state as its correlated SUC machine). In particular, this holds for the environment machine, and thus its output is identical in both executions for all inputs (n, z). This completes the proof of the lemma. ■

Lemma 3.15 *Let $\pi, \pi', \mathcal{S}, \mathcal{S}', \mathcal{Z}, \mathcal{Z}'$ be as above. Then for every $d \in \mathbb{N}$:*

$$\left\{\text{SUC-IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(n,z)\right\}_{n\in\mathbb{N}, z\in\cup_{\kappa\leq n^d}\{0,1\}^\kappa} \equiv \left\{\text{IDEAL}_{\mathcal{F}',\mathcal{S}',\mathcal{Z}'}(n,z)\right\}_{n\in\mathbb{N}, z\in\cup_{\kappa\leq n^d}\{0,1\}^\kappa}$$

Proof Sketch: The proof is the same as above, with the exception that instead of $\mathcal{A}$ being constructed from $\mathcal{A}'$ as in Lemma 3.14, $\mathcal{S}'$ is constructed from $\mathcal{S}$ via the transformation $T_{\mathcal{S}}$. This transformation preserves the same behavior of the ideal adversary by simply rerouting messages via the appropriate tapes. In addition, when $\mathcal{S}$ corrupts a party by sending a corruption message to it in the SUC model, $\mathcal{S}' = T_{\mathcal{S}}(\mathcal{S})$ sends the corruption message unchanged to $\mathcal{F}'$. This is dealt with in $\mathcal{F}'$ according to the standard corruption procedure which has the same effect as in the SUC execution (note that $\mathcal{S}'$ does not allow $\mathcal{S}$ to notify the environment of the corruption since in the UC model $\mathcal{F}'$ notifies the party who in turn notifies the environment; nevertheless, the result is the same). Everything else remains the same and so the same inductive proof holds. We conclude that the environment's output bit is identical for all inputs (n, z) in both executions. ■

We are now ready to complete the proof of the main theorem. Combining Eq. (2) with Lemmas 3.14 and 3.15, we have:

$$\begin{aligned} n^c &\leq \left|\Pr[\text{IDEAL}_{\mathcal{F}',\mathcal{S}',\mathcal{Z}'}(n,z)=1] - \Pr[\text{HYBRID}^{\mathcal{G}',\mathcal{F}_{\text{AUTH}}}_{\pi',\mathcal{A}',\mathcal{Z}'}(n,z)=1]\right| \\ &= \left|\Pr[\text{SUC-IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(n,z)=1] - \Pr[\text{SUC-HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}(n,z)=1]\right| \end{aligned}$$

Therefore, for an infinite number of pairs (n, z):

$$\left|\Pr[\text{SUC-IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(n,z)=1] - \Pr[\text{SUC-HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}(n,z)=1]\right| \geq n^c$$

in contradiction to the assumption that π SUC-securely realizes $\mathcal{F}$. Therefore π' UC-securely realizes $\mathcal{F}'$. ■

4 UC to SUC Security and SUC Composition

In order to achieve SUC composition, we still need to prove that the security of a UC protocol that is the result of the transformation from the SUC setting implies the SUC security of the original SUC protocol. This is because this allows us to apply the UC composition theorem to obtain a new UC secure protocol, whose SUC security we can derive from this proof. This proof is in the opposite direction of the previous one and is very similar; however, the required transformations are different.

4.1 Required Transformations

Definition 4.1 *We define the* adversary transformation $T_{\mathcal{A}} : \{0,1\}^* \to \{0,1\}^*$*, in the following manner. $T_{\mathcal{A}}(\cdot)$ is given a code tape $C_{\mathcal{A}}$ of adversarial machine $\mathcal{A}$ in the SUC model, and generates the code $C_{\mathcal{A}'}$ of machine $\mathcal{A}'$ in the full UC model, as follows:*

- *Upon receiving an incoming public header (a message with no private content) on the incoming communication tape, $T_{\mathcal{A}}(C_{\mathcal{A}})$ processes the header with the function in $C_{\mathcal{A}}$ that processes incoming public headers from the router.*
- *When $C_{\mathcal{A}}$ wishes to instruct the router to deliver a message m with header $H_{\mathcal{F}}(m)$, $T_{\mathcal{A}}(C_{\mathcal{A}})$ writes the message $(\mathsf{output}, sid, H_{\mathcal{F}}(m))$ on the incoming communication tape of the functionality $\mathcal{F}_{\text{AUTH}}$ instead.*
- *When $C_{\mathcal{A}}$ wishes to write a message on the incoming communication tape of an ideal functionality $\mathcal{G}$, $T_{\mathcal{A}}(C_{\mathcal{A}})$ writes the message on the incoming communication tape of the corresponding functionality $\mathcal{G}'$ instead.*
- *When $C_{\mathcal{A}}$ wishes to corrupt a protocol party, $T_{\mathcal{A}}(C_{\mathcal{A}})$ passes the corruption message to the protocol party directly. We note that the message from $C_{\mathcal{A}}$ to the environment in the SUC model is not passed by $T_{\mathcal{A}}(C_{\mathcal{A}})$ to the environment in the UC model (a different mechanism is used for this purpose in the UC model).*
- *When $C_{\mathcal{A}}$ wishes to write a message for the environment on the outgoing communication tape, $T_{\mathcal{A}}(C_{\mathcal{A}})$ writes the message on the output tape instead.*

The next lemma is proved identically to Lemma 3.9.

Lemma 4.2 *Let $\mathcal{A} \in PPT^{SUC}$ be the adversary in a* SUC-REAL *(resp.,* SUC-HYBRID*) execution. Then, machine $T_{\mathcal{A}}(\mathcal{A})$ is PPT in the* REAL *(resp.,* HYBRID*) execution of the UC model.*

The environment transformation is defined in a similar manner to T_P. The transformation requires the polynomials $q, q_1, \ldots, q_m$, where $q(\cdot)$ is the polynomial that bounds the runtime of the SUC environment $\mathcal{Z}$ and $q_1, \ldots, q_m$ are the polynomials that bound the runtime of the SUC protocol parties.

Definition 4.3 *We define the* environment transformation $T_{\mathcal{Z}} : \{0,1\}^* \times \mathbb{Z}[x]^{m+1} \to \{0,1\}^*$, *in the following manner. $T_{\mathcal{Z}}(\cdot)$ is given a code tape $C_{\mathcal{Z}}$ of the environment machine $\mathcal{Z}$ in the SUC model and polynomials $q, q_1, \ldots, q_m$, and generates the code $C_{\mathcal{Z}'}$ of machine $\mathcal{Z}'$ in the full UC model, as follows:*

- *Upon execution with input (n, z'), $\mathcal{Z}'$ first verifies the input:*
  - *If $z' = (1^k, 0, z)$ where $k > 1$ and z is an arbitrary string, and k satisfies the equation $k = q(n+|z|) + \sum_{i=1}^{m} q_i\left(n + q^2(n+|z|)\right)$, then $\mathcal{Z}'$ invokes $\mathcal{Z}$ internally with input (n, z).*
  - *Otherwise, $\mathcal{Z}'$ outputs 0 and halts.*
- *Upon receiving a message from the adversary via its subroutine output tape connected to $\mathcal{A}'$, $\mathcal{Z}$ writes the message on the simulated incoming communication tape of $\mathcal{Z}$.*
- *When $\mathcal{Z}$ generates its output bit, $\mathcal{Z}'$ outputs the same bit and halts.*
- *When $\mathcal{Z}$ writes the ith (reactive) input x_i on the simulated input tape of party P_j, $\mathcal{Z}'$ passes the input $x'_i = (1^{k_i}, 0, x_i)$ to P'_j where k_i (called the token extension) satisfies $\sum_{\ell=1}^{i} k_\ell = q_j(\sum_{\ell=1}^{i} |x_\ell| + n)$ and $k_1, \ldots, k_{i-1}$ are the previous token extensions passed to P_j.*

- *When $\mathcal{Z}$ writes the $HALT$ input on the simulated input tape of party P_j, $\mathcal{Z}'$ externally writes the input $(1^k, 0, x)$ for (k,x) that does not satisfy the condition on the token extensions.*

- *When $\mathcal{Z}$ writes an input on the simulated incoming communication tape of the adversary, $\mathcal{Z}'$ writes the same input on the adversary's input tape.*

Lemma 4.4 *If machine $\mathcal{Z} \in PPT^{SUC}$ acts as the environment machine in an SUC execution, then machine $\mathcal{Z}' \in ITM^{UC}$ that runs the code $T_{\mathcal{Z}}(\mathcal{Z})$ is locally p-bounded for some polynomial $p(\cdot)$ in the respective full UC execution.*

Proof Sketch: This can be proved in an almost identical way to the proof of Lemma 3.2. Observe that the "shell" of $\mathcal{Z}'$ performs a similar input check to verify that $\mathcal{Z}'$ has enough time to run. If the padded input (n, z') does not contain enough tokens $\mathcal{Z}'$ halts. Otherwise, $\mathcal{Z}'$ extracts the effective auxiliary input z and runs the original code of $\mathcal{Z}$ on the input (n, z). Since the original $\mathcal{Z}$ runs at most $q(n + |z|)$ steps we have (initially) $n_O \leq q(n + |z|)$ and thus k must be at least $q(n + |z|)$. Observe that the transformation changes the core behaviour of $\mathcal{Z}'$ (the original code) to pad each reactive input x_i to a protocol party P_j with k_i tokens such that $\sum_{\ell=1}^{i} k_\ell = q_j(n + \sum_{\ell=1}^{i} |x_\ell|)$, thus increasing n_O by an additional $\sum_{j=1}^{m} q_j(n + \sum_{i=1}^{t} |x_i|)$ where t is the maximum number of reactive inputs passed to a party (recall that q_j is the polynomial that bounds the running time of the jth SUC protocol party). Observe that since for every i, $|x_i| \leq q(n + |z|)$ and $t \leq q(n + |z|)$ we have:

$$q_j\left(n + \sum_{i=1}^{t} |x_i|\right) \leq q_j\left(n + \sum_{i=1}^{t} q(n + |z|)\right) \leq q_j\left(n + \sum_{i=1}^{q(n+|z|)} q(n + |z|)\right) = q_j\left(n + q^2(n + |z|)\right)$$

Thus $\mathcal{Z}'$ must receive $k = q(n + |z|) + \sum_{i=1}^{m} q_i\left(n + q^2(n + |z|)\right) = poly(n + |z|)$ tokens, implying that

$$\tilde{n} = n_I - n_O = n + |z| + k - n_O \geq n + |z|.$$

We now bound the runtime of $\mathcal{Z}'$. The shell of $\mathcal{Z}'$ runs at most $p_1(n_I) = p_1(n + |z| + k)$ steps for some polynomial $p_1(\cdot)$, and the modified core of $\mathcal{Z}'$ runs at most $q^2(n + |z|) + p_2(n + |z| + k)$ for some polynomial $p_2(\cdot)$ (quadratic in the number of steps of the unmodified core and another polynomial of the all the padding required for the protocol parties). Since $\mathsf{steps}(\mathcal{Z}') = p_1(n + |z| + k) + q^2(n + |z|) + p_2(n + |z| + k) = poly(n + |z|)$ we conclude that $\mathcal{Z}'$ runs in time $p(n + |z|)$ for some polynomial p, thus $\mathcal{Z}'$ is p-bounded. ■

Definition 4.5 *We define the ideal simulator reverse transformation $B_{\mathcal{S}} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, in the following manner. $B_{\mathcal{S}}(\cdot)$ is given a code tape $C_{\mathcal{S}'}$ of a UC ideal adversary machine $\mathcal{S}'$ and the code tape $C_{control}$ of a UC control function machine, and generates the code $C_{\mathcal{S}}$ of machine $\mathcal{S} \in ITM^{SUC}$. $\mathcal{S}$ internally invokes an instance of $\mathcal{S}'$ and then works as follows:*

- *Upon receiving a message from the environment $\mathcal{Z}$ on its incoming communication tape, $\mathcal{S}$ writes the message on $\mathcal{S}'$'s input tape.*

- *When $\mathcal{S}'$ writes a value to its output tape, $\mathcal{S}$ sends it to the environment on its outgoing communication tape.*

- *When $\mathcal{S}$ receives an incoming message from the ideal functionality $\mathcal{F}'$, $\mathcal{S}$ writes the message on the simulated incoming communication tape of $\mathcal{S}'$.*

- *When $\mathcal{S}'$ performs an external write to a machine $\mu = (id, \hat{M})$:*
  - *If $C_{control}$ disallows the message, the instruction is ignored.*
  - *If id is the designated identifier of an existing machine M and $C_{\hat{M}} = C_M$ then $\mathcal{S}$ treats it as an outgoing message (see below).*
  - *If id is the designated identifier of an existing machine M and $C_{\hat{M}} \neq C_M$, then $\mathcal{S}$ simulates $\mathcal{S}'$ receiving a special error message from the control function.*
  - *If id does not refer to an existing machine, $\mathcal{S}$ internally simulates for $\mathcal{S}'$ an invocation of the machine μ.*
- *When $\mathcal{S}'$ writes an outgoing message to an already existing machine (using external-write):*
  - *If the message is a corruption message of the form $(\mathsf{corrupt}, id, p)$, where p is the corruption parameter, $\mathcal{S}$ sends the message $(\mathsf{corrupt}, id)$ to party id via the router and also sends it to the environment.*
  - *If the message is written on the simulated incoming communication tape of a protocol party, $\mathcal{S}$ ignores the message (in the UC model $\mathcal{S}$ can write directly to a party, but this is not true in the SUC model; thus such messages are just ignored).*
  - *If the message is written on the simulated incoming communication tape of an ideal functionality $\mathcal{F}'$, $\mathcal{S}$ sends the message to the corresponding ideal functionality $\mathcal{F}$ in the SUC execution.*

The generated machine $\mathcal{S}$ is called a black-box shell *of ideal adversary $\mathcal{S}' \in PPT^{UC}$.*

The next lemma is proved identically to Lemma 3.12.

Lemma 4.6 *If $\mathcal{S}' \in PPT^{UC}$ then the corresponding black box shell $\mathcal{S}$ is PPT in the SUC environment.*

4.2 Proof of Security

We now formally state the security theorem in the opposite direction.

Theorem 4.7 *Let π be an SUC protocol running with up to m parties and let $\mathcal{F}$ be an ideal functionality. If $T_P(\pi)$* UC-realizes *$\phi(F)$ in the $(\phi(\mathcal{G}), \mathcal{F}_{\text{AUTH}})$-hybrid model, then π* SUC-securely computes *$\mathcal{F}$ in the $\mathcal{G}$-hybrid model.*

Proof Sketch: The proof is almost identical to that of Theorem 3.13 and is the details are therefore omitted. The main difference is that we change the input of the environment, and therefore we show that the output distribution when $\mathcal{Z}$ in the SUC model is run with (n, z) is identical to $\mathcal{Z}'$ in the UC model with a different, associated input (n, z'). This suffices since if security does not hold for infinitely many inputs of the form (n, z) in the SUC model, then it does not hold for infinitely many inputs of the form (n, z') in the UC model, contradicting the assumed security.

In this proof we work in the opposite direction. That is, we start with an SUC adversary $\mathcal{A}$, and derive a UC adversary $\mathcal{A}' = T_{\mathcal{A}}(\mathcal{A})$. From the security of π' in the full UC model, we derive an ideal simulator $\mathcal{S}'$ and then construct a black box shell $\mathcal{S}$ of $\mathcal{S}'$. We then show that if there exists an environment $\mathcal{Z}$ in the SUC model that distinguishes with non-negligible probability, then the environment $\mathcal{Z}' = T_{\mathcal{Z}}(\mathcal{Z})$ distinguishes with non-negligible probability in the UC model, thus contradicting the security of π'. Thus, the steps are the same as in the proof of Theorem 3.13 but in the opposite direction. $\blacksquare$

4.3 Achieving SUC Composition

In this section, we prove Corollary 2.4 as a corollary of Theorems 3.13 and 4.7. Recall that Corollary 2.4 states that concurrent general composition holds in the SUC model.

Corollary 4.8 (Corollary 2.4 – restated:) *Let π be a protocol that SUC-securely computes a functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. If protocol ρ SUC-securely computes $\mathcal{G}$ in the real (resp., $\mathcal{H}$-hybrid) model, then π^ρ SUC-securely computes $\mathcal{F}$ in the real (resp., $\mathcal{H}$-hybrid) model.*

Proof: In order to simplify notation, we prove the corollary in the real model; the proof in the $\mathcal{H}$-hybrid model is almost identical. Let π, ρ, $\mathcal{F}$ and $\mathcal{G}$ be as in the theorem, and let $\pi' = T_P(\pi)$, $\rho' = T_P(\rho)$, $\mathcal{F}' = \phi(\mathcal{F})$ and $\mathcal{G}' = \phi(\mathcal{G})$ be the corresponding transformations to the full UC framework.

By Theorem 3.13, we have that π' UC-realizes $\mathcal{F}'$ in the $(\mathcal{G}', \mathcal{F}_{\text{AUTH}})$-hybrid model, and ρ' UC-realizes $\mathcal{G}'$ in the $\mathcal{F}_{\text{AUTH}}$-hybrid model. Thus, by the UC composition theorem, we have that $(\pi')^{\rho'}$ UC-realizes $\mathcal{F}'$ in the $\mathcal{F}_{\text{AUTH}}$-hybrid model.

Next, observe that for every adversary $\mathcal{S}'$ interacting with an environment and parties running $(\pi')^{\rho'}$ there exists an adversary $\mathcal{S}''$ interacting with the same environment and with parties running the protocol $(\pi^\rho)' = T_P(\pi^\rho)$ such that the output of the environment is *identical* in both cases. This is achieved by just having the adversary $\mathcal{S}''$ reroute the messages for subroutine ρ' of π' to the main machine $(\pi^\rho)'$. We therefore have that $(\pi^\rho)'$ UC-realizes $\mathcal{F}'$ in the $\mathcal{F}_{\text{AUTH}}$-hybrid model

Finally, applying Theorem 4.7, we conclude that π^ρ SUC-securely computes $\mathcal{F}$ in the real model, in the SUC setting. This completes the proof. $\blacksquare$

5 An Illustrative Example – Proving in the UC vs. SUC Models

In this section, we demonstrate the difference between proving security in the full UC framework and in the SUC framework. We consider the classic commitment functionality $\mathcal{F}_{\mathrm{COM}}$, due to its relative simplicity. We also consider realizing the $\mathcal{F}_{\mathrm{ZK}}$ functionality in the $\mathcal{F}_{\mathrm{COM}}$-hybrid model, since existing protocols "gloss over" the details of using the composition theorem correctly.

5.1 Differences in Describing the Ideal Functionality for Commitments

Before describing the functionality, we need to introduce the *delayed output* terminology, which is a convention that appears in the full UC framework. Quoting from [6, Sec. 6.2]:

> More precisely, we say that an ideal functionality $\mathcal{F}$ sends a $\mathsf{delayed\ output}$ v to party P if it engages in the following interaction: Instead of simply outputting v to P, $\mathcal{F}$ first sends to the adversary a message that it is ready to generate an output to P. If the output is public, then the value v is included in the message to the adversary. If the output is private then v is not mentioned in this message. Furthermore, the message contains a unique identifier that distinguishes it from all other messages sent by $\mathcal{F}$ to the adversary in this execution. When the adversary replies to the message (say, by echoing the unique identifier), $\mathcal{F}$ outputs the value v to P.

We now consider the definition of secure commitments. For simplicity, we consider the *single* commitment functionality (typically, the multiple commitment functionality is used, but this even further complicates the definition). This is the definition that appears in [5, Sec. 7.3.1]:

FIGURE 5.1 (Functionality $\mathcal{F}_{\mathrm{COM}}$ for the Full UC Framework)

1. Upon receiving an input $(\mathsf{Commit}, sid, x)$ from C, verify that $sid = (C, R, sid')$ for some R, else ignore the input. Next, record x and generate a public delayed output $(\mathsf{Receipt}, sid)$ to R. Once x is recorded, ignore any subsequent Commit inputs.
2. Upon receiving an input (Open, sid) from C, proceed as follows: If there is a recorded value x then generate a public delayed output (Open, sid, x) to R. Otherwise, do nothing.
3. Upon receiving a message $(\mathsf{Corrupt\text{-}committer}, sid)$ from the adversary, output a $\mathsf{Corrupted}$ value to C, and send x to the adversary. Furthermore, if the adversary now provides a value x', and the $\mathsf{Receipt}$ output was not yet written on R's tape, then change the recorded value to x'.

The Ideal Commitment Functionality $\mathcal{F}_{\mathrm{COM}}$

In contrast, in the SUC framework the functionality description is far simpler. Before writing the functionality, we introduce a convention that was used in [11] for the public headers and private contents in functionalities. The "operation labels" (e.g., Commit, $\mathsf{Receipt}$, etc.) and the session identifiers are by convention (and unless explicitly stated otherwise) part of the public header, and the rest of the message constitutes the private contents. We have:

FIGURE 5.2 (Functionality $\mathcal{F}_{\text{COM}}$ for the SUC Framework)

1. Upon receiving an input $(\mathsf{Commit}, sid, x)$ from C, verify that $sid = (C, R, sid')$ for some R, else ignore the input. Next, record x and send $(\mathsf{Receipt}, sid)$ to R. Once x is recorded, ignore any subsequent Commit inputs.
2. Upon receiving an input (Open, sid) from C, proceed as follows: If there is a recorded value x then send (Open, sid, x) to R. Otherwise, do nothing.

The Ideal Commitment Functionality $\mathcal{F}_{\text{COM}}$

Explaining the differences between the functionalities. In the full UC framework, it is necessary to refer to public delayed outputs, since honest parties write their inputs locally to ideal functionalities; to be more exact, an ideal call is a subroutine invocation. Thus, in interactive scenarios, it is necessary for the ideal functionality to explicitly communicate with the adversary to ask permission to send the receipt, and so on. Due to the fact that this is tiresome to describe each time, the convention of a "delayed output" was introduced. In contrast, in the SUC framework, since the adversary automatically controls all delivery, it suffices to naturally send messages. However, this does come at the price of explicitly stating which parts of the messages are public (and seen by the adversary when it delivers) and which parts are private. Nevertheless, by our convention, this is typically simple.

A more significant difference arises in the context of corruption. In the full UC model, an ideal functionality is modeled as a *subroutine* of the main protocol instance. Therefore, parties "send" messages/inputs to an ideal functionality $\mathcal{F}$ by writing them directly on the input tape of $\mathcal{F}$. This means that the adversary cannot change the contents of such a written message, even in the case that the party is corrupted before the input was effectively used. In real protocols, it is often possible for the adversary to make such a change. (For example, consider the case that the honest party sends its first message and is corrupted before it is delivered. In this case, the adversary can choose not to deliver that message and instead send a new message in its place for the corrupted party, possibly using a different input. Thus, this has to also be possible in the ideal model.) This forces such treatment to be explicitly defined in the ideal functionality. In contrast, in the SUC framework, this issue does not arise at all. This is because all messages, *including inputs to an ideal functionality and messages in a real protocol*, are treated in the same way and sent via the router. By the way the router is defined, an adversary can choose not to deliver messages to an ideal functionality in the same way that it can choose not to deliver messages in a real protocol.

5.2 Proving Security of Commitment Protocols and Zero Knowledge Protocols

In this section, we consider the problem of constructing UC commitments in the CRS model, and then zero knowledge protocols using UC commitments. This is the standard way of working; see [9, 11], and see [13] for a more recent work following the same paradigm. The authors of [13] claim security of their zero knowledge protocol by referring to the proof of security of zero knowledge from commitments that appears in [9]. However, this proof is much closer to the SUC framework and does not take into account a number of issues that must be considered in the (current version of the) full UC model. We describe *some* of the additional issues that need to be taken into account in order to prove the full UC security of the zero knowledge protocol from full UC commitments. For the sake of concreteness, when considering polynomial time, we refer specifically to the constructions in [13].

Before proceeding, denote the commitment protocol of [13] by $\Pi_{\rm COM}$, the CRS functionality by $\mathcal{F}_{\rm CRS}$, and the zero knowledge protocol of [9, 13] by $\Pi_{\rm ZK}$. Protocol $\Pi_{\rm ZK}$ works by running the classic zero knowledge Hamiltonicity protocol of Blum [2], while using UC commitments. Actually, since many commitments are needed with respect to the same CRS, the multiple commitment functionality $\mathcal{F}_{\rm MCOM}$ is used but for simplicity we will ignore this here. Note that the commitment protocol $\Pi_{\rm COM}$ in [13] uses a fully-homomorphic encryption scheme denote $\rm Q_{ENC}$ and a CCA-secure encryption scheme $\rm ENC_{CCA}$.

Proof of polynomial-time. One of the requirements of the UC composition theorem is that all the protocols involved are polynomial time. The mentioned proofs do not formally prove that the protocols are polynomial time. In the SUC model, the fact that $\Pi_{\rm COM}$ in [13] is polynomial time is immediate, and simply follows from the fact that the $\rm Q_{ENC}$ and $\rm ENC_{CCA}$ encryption schemes run in polynomial time (since in each invocation each party trivially runs in time that is polynomial in the security parameter and input; see Section 2.1 for why this suffices in the SUC framework). However, in order to prove that $\Pi_{\rm COM}$ in [13] is polynomial time in the full UC framework, one needs to first pad the input of each party in $\Pi_{\rm COM}$ with sufficient tokens, so that it runs in time that is polynomial in the length of its (padded) input minus the length of the inputs/messages that it sends to $\mathcal{F}_{\rm CRS}$. If $\mathcal{F}_{\rm CRS}$ is assumed to be a local functionality (e.g., secure setup), then this is not difficult since the only input to $\mathcal{F}_{\rm CRS}$ is the pair (CRS, sid). However, if $\mathcal{F}_{\rm CRS}$ is implemented via coin-tossing using a local $\mathcal{F}_{\rm CRS}$ functionality (as suggested in the JUC [12] solution to achieving independent CRS invocations per protocol), then the number of tokens needed to be provided is different. Essentially, a different $\mathcal{F}_{\rm CRS}$ ideal functionality has to be defined for each of these cases. (The reason that a different ideal functionality is needed is that the functionality defines the length of the input, which depends on the number of tokens needed.)

Consider next the case of constructing $\Pi_{\rm ZK}$ using $\mathcal{F}_{\rm COM}$. These zero-knowledge protocols make multiple calls to the commitment functionality. The number of calls to $\mathcal{F}_{\rm COM}$, and thus the length of the input written by the parties in $\Pi_{\rm ZK}$ to $\mathcal{F}_{\rm COM}$, differs *significantly* when the zero-knowledge is based on Hamiltonicity versus when it is based on 3 coloring. The proof of polynomial-time complexity must take into account that for Hamiltonicity, for a graph with n nodes, $O(n^3)$ calls to $\mathcal{F}_{\rm COM}$ are made (repeating n times where in each time a matrix of size $O(n^2)$ is committed to). However, the size of the graph depends on the Karp reduction of the statement being proven to Hamiltonicity, and this must also be counted. This bound must then be included in the ideal functionality for $\mathcal{F}_{\rm COM}$, since the actual length of the input includes these tokens. Notice, however, that the number of tokens needed in 3 coloring will be different, and so the definition of $\mathcal{F}_{\rm COM}$ can actually depend on the implementation of $\mathcal{F}_{\rm ZK}$ as used by $\Pi_{\rm COM}$. To make this even more complex, if $\mathcal{F}_{\rm COM}$ uses $\mathcal{F}_{\rm CRS}$ as described above, then the number of token further depends on whether $\mathcal{F}_{\rm CRS}$ is a local functionality or derived by some type of coin-tossing protocol.

We are not aware of any research paper whose focus is protocol construction that relates to the issue of defining the number of tokens–equivalently how much to pad the input–when defining the functionality, and proving that the protocol is polynomial time as defined in the full UC framework.

Subroutine Respecting Protocols. The UC composition theorem demands that protocols are subroutine respecting; see [6]. Informally speaking, this means that subroutines only accept messages from other parties or subsidiaries of the subroutine instance. In addition, upon the first activation, the adversary receives notification of the code and SID of the instance. Since these are messages sent to the adversary, they need to be dealt with by the adversary in the proof of security. To the best of our knowledge, the adversary's treatment of these notifications are typically not described.

Corruptions. In the full UC framework, the protocol specification has to include what the parties should do upon receiving a Corrupt message. This is due to the fact that the UC framework enables great flexibility in dealing with corruptions (and thus can model partial corruptions, proactive corruptions, and so on). In contrast, in the SUC model, a party is either honest or fully corrupted, and in the latter case the adversary obtains full control of the party. Although describing what a party should do upon corruption is not complicated, it is once again an example of a detail that needs to be addressed, but is to the best of our knowledge omitted in current protocol specifications.

Order of activations. In the full UC framework, the order of activations depends on the adversary and on the protocol, and is derived from the order of external write calls made by the machines in the system. Each machine can only write one external message (be it input to a subroutine, output, or a regular message) per activation, and by writing the message it passes the execution to the receiving machine. This means that multiple invocation patterns are possible, yielding multiple case analyses in the proof. In addition, when writing the proof, one must distinguish between the different types of messages (writing to an ideal functionality is fundamentally different to sending a message to another party). Both of these complicate the presentation and make it harder for one writing the proof to be exact. In contrast, in the SUC model, one of our aims was to make the order of activations the same in all models (real, ideal and hybrid) and to use the same method for all types of messages. (The only exception is the parties' inputs written by the environment and their outputs read by the environment.) Thus, the scheduling of activations and the terminology with respect to messages is always the same (under full control of the adversary), simplifying the presentation.

Conclusions – current UC research and UC/SUC proofs. We are not aware of *any written proof* in the UC framework that actually takes these details into account. Rather, researchers writing protocols in the UC framework do not specify the number of tokens needed in order to be polynomial time (which is the most serious issue), do not describe what the adversary should do with invocation messages, do not consider the varying order of activations, and so on. Essentially, researchers today write their proofs as if they are working in something similar to the SUC framework. The main contribution of this paper can therefore be viewed as a *justification of the soundness* of working in this way. In addition, we provide an *exact* model that can be used, instead of handwaving away the full UC details. Finally, our proof that SUC protocols are actually UC secure (with the appropriate adjustments) means that for the standard interactive secure computation tasks, nothing is lost by working with our simpler model.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments, especially for their suggestion to write Section 5.

References

[1] D. Beaver. Foundations of Secure Interactive Computing. In *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 377–391, 1991.

[2] M. Blum. How to Prove a Theorem So No One Else Can Claim It. *Proceedings of the International Congress of Mathematicians*, pages 1444–1451, USA.

[3] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. In the *Journal of Cryptology*, 13(1):143–202, 2000.

[4] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In the 42*nd FOCS*, pages 136–145, 2001.

[5] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. *Cryptology ePrint Archive*, Report 2000/067, revision of 13 December 2005.

[6] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. *Cryptology ePrint Archive*, Report 2000/067, revision of 16 July 2013.

[7] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. *Cryptology ePrint Archive*, Report 2000/067, revised 13 Dec. 2005, and re-revised April 2013.

[8] R. Canetti. Obtaining Universally Compoable Security: Towards the Bare Bones of Trust. In *ASIACRYPT 2007*, pages 88–112, 2007.

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

[10] R. Canetti and A. Herzberg. Maintaining Security in the Presence of Transient Faults. In *CRYPTO 1994*, Springer (LNCS 839), pages 425–438, 1994.

[11] R. Canetti, Y. Lindell, R. Ostrovsky and A. Sahai. Universally Composable Two-Party and Multi-Party Secure Computation. In the 34*th STOC*, pages 494–503, 2002. Reference is to page 13 of *Cryptology ePrint Archive Report 2002/140*, version of 14 July 2003.

[12] R. Canetti and T. Rabin. Universal Composition with Joint State. In *CRYPTO 2003*, Springer-Verlag (LNCS 2729), pages 265–281, 2003.

[13] I. Damgård, A. Polychroniadou and V. Rao. Adaptively Secure UC Constant Round Multi-Party Computation. *Cryptology ePrint Archive*, Report 2014/830, 2014.

[14] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications.* Cambridge University Press, 2004.

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

[16] D. Hofheinz and J. Müler-Quade and R. Steinwandt. Initiator-Resilient Universally Composable Key Exchange. *ESORICS,* 2003. Extended version at the eprint archive, eprint.iacr.org/2003/063.

[17] D. Hofheinz, J. Müller-Quade and D. Unruh. Polynomial Runtime and Composability. *IACR Cryptology ePrint Archive*, report 2009/23, 2009.

[18] D. Hofheinz and V. Shoup. GNUC: A New Universal Composability Framework. *IACR Cryptology ePrint Archive*, report 2011/303, 2011.

[19] J. Katz, U. Maurer, B. Tackmann, V. Zikas. Universally Composable Synchronous Computation. *Theory of Cryptology Conference (TCC)* 2013: 477-498.

[20] R. Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. *CSFW 2006,* pp. 309-320.

[21] R. Küsters and M. Tuengerthal. The IITM Model: a Simple and Expressive Model for Universal Composability. *IACR Cryptology ePrint Archive*, report 2013/25, 2013.

[22] Y. Lindell. General Composition and Universal Composability in Secure Multi-Party Computation. In the *Journal of Cryptology,* 22(3):395-428, 2009. An extended abstract appeared in the 44*th FOCS*, pages 394–403, 2003.

[23] N. Lynch, R. Segala and F. Vaandrager. Compositionality for Probabilistic Automata. *14th CONCUR,* Springer (LNCS 2761), pages 208-221, 2003. Fuller version appears in MIT Technical Report MIT-LCS-TR-907.

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

[25] B. Pfitzmann and M. Waidner. Composition and Integrity Preservation of Secure Reactive Systems. In *7th ACM Conference on Computer and Communication Security*, pages 245–254, 2000.