A Simpler Variant of Universally Composable Security for Standard Multiparty Computation[⋆]

Ran Canetti[1], Asaf Cohen[2], and Yehuda Lindell[2]

[1] Boston University and Tel-Aviv University.
`canetti@tau.ac.il`
[2] Bar-Ilan University, Israel.
`asafc@me.com,lindell@biu.ac.il`

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, 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 that for every protocol that can be represented in the SUC framework, the protocol is secure in SUC if and only if it is secure in UC. Therefore, the UC composition theorem holds and any protocol that is proven secure under SUC is secure under the general framework (with some technical changes to the functionality definition). As a result, protocols that are secure in the SUC framework are secure when an a priori unbounded number of concurrent executions of the protocols take place (relative to the same fixed set of parties).

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

[⋆] The full version of this work can be found on the *IACR Cryptology ePrint Archive* [9]. This work was supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 189/11).

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). Unfortunately, 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 is automatically 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. In Section 3, we provide an illustrative example demonstrating that it is significantly more simple to work in the SUC model than in the full UC model.

Remark: We assume familiarity with the ideal/real model paradigm and the standard definitions of security for multiparty computation; see [3] and [15, Chapter 7] for a detailed treatment on these definitions. In addition, we assume that the reader has basic familiarity and understanding of the notion of UC security. This paper is not intended as a tutorial of the UC framework.

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 this definition of security are called universally composable.

As in other general definitions (e.g., [16, 25, 1, 27, 3, 15]), 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 [18, 19, 22].

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, 15]), with the addition of an interactive environment as is required for proving concurrent general composition [23]. In this section we outline the SUC definition, and discuss the main differences between it and the full UC framework.

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], unlike the modeling in the full UC framework where subprotocols are invoked as separate ITMs.

The Main Differences Between UC and SUC

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, 15]. We therefore adopt authenticated channels as the default in SUC, thus simplifying the description of protocols (formally, 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].

We stress that although the number of parties in an SUC protocol is a priori fixed, security is guaranteed under composition even when an *unbounded* number of instances of the protocol are run concurrently. This is obtained via the SUC/UC composition theorem.

Expressibility. As we have mentioned, there are cryptographic tasks that can be modeled in the UC framework, but not in the SUC framework. One class of examples is non-interactive cryptographic primitives like digital signatures, encryption, pseudorandom functions and so on. These cannot be modeled in the SUC framework since any interaction with an ideal functionality requires communication that goes via the router and thus its scheduling is controlled by the adversary. This does not model the real-world behavior of local computation for these primitives. Another example is that of protocols in synchronous networks that guarantee output to all parties. This is not possible since the adversary controls the scheduling and thus it is inherently asynchronous. In addition, the adversary can always block messages. Despite this, the SUC framework suffices for modeling any interactive protocol between parties in the most common model of communication for the concurrent setting where the adversary has full control over all message scheduling.

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 *interface* to the UC framework that offers the same security standard, and can simplify the process of proving UC security of protocols.

Organization. Due to lack of space in this extended abstract, the proof of equivalence between UC and SUC security is not included, and can be found in the full version [9]. Although this proof is crucial to this work, our main contribution is a simple model that can be used. As such, a presentation of the framework, and a demonstration of why it is easier to use – as can be found in Section 3 – covers the main goals.

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, 18, 20]. In addition, there have been other frameworks developed to capture the same setting of dynamic concurrency as that of the UC framework [24, 21, 19, 22, 27]. 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 and our aim is to capture concurrency for more restricted tasks and obtain a simpler definition. Due to its simplicity, our work can also act as a bridge for connecting the full UC framework with alternative formalisms like [26]. A similar attempt at providing a simplified framework, but without a proof of equivalence, also appeared in [28, Ch. 4].

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 its inputs.

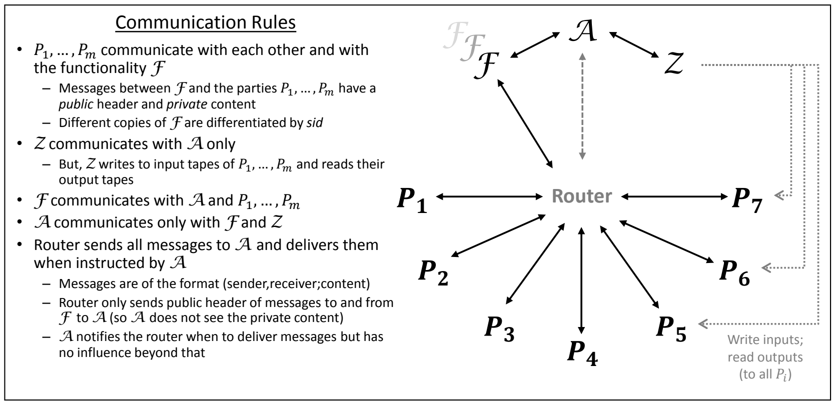
It is important to note that even if the inputs are short (e.g., constant length), a polynomial-time party can still run in time that is polynomial in the security parameter in every invocation. In order to see this, observe that $\left(n + \sum_{j=1}^{\ell} |x_j|\right)^2 > \sum_{j=1}^{\ell} n \cdot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

Interactive Turing machines. The formal definition of interactive Turing machines (ITMs) can be found in the full version.

2.2 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 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

Fig. 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 and known to all.[3]

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.[4] 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.[5] We remark that the public headers of different messages in an execution must be *different*, so that there is no ambiguity regarding the adversary's instructions to the router (formally, the router ignores any new message that has an identical public header to a previously sent different message).

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

[3] 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.

[4] 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.

[5] 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.

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, 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 (e.g., as is used to model digital signatures 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 2.3 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 "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. All machines are initialized to have the same value 1^n on their security parameter tapes. Then, 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.

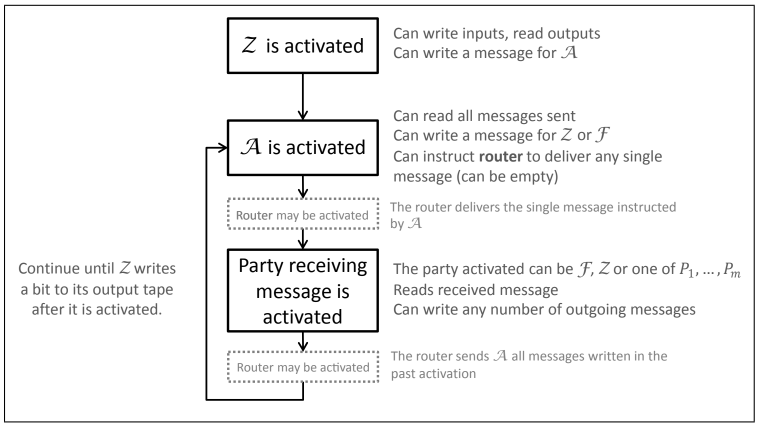

$\mathcal{Z}$ is activated
Can write inputs, read outputs
Can write a message for $\mathcal{A}$
$\mathcal{A}$ is activated
Can read all messages sent
Can write a message for $\mathcal{Z}$ or $\mathcal{F}$
Can instruct **router** to deliver any single message (can be empty)
Router may be activated
The router delivers the single message instructed by $\mathcal{A}$
Continue until $\mathcal{Z}$ writes a bit to its output tape after it is activated.
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

Fig. 2. The execution flow and order of activations

2.3 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, by convention, the adversary is required to send 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). For the same reason, it is not possible to model *proactive security* in the SUC framework [11].

2.4 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 $\text{SUC-REAL}_{\pi,\mathcal{A},\mathcal{Z}}(n, z)$, where z is the input to $\mathcal{Z}$.
– 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.[6]

– 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.[7] 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 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.

2.5 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 $\textsc{suc-ideal}$ and $\textsc{suc-real}$ notation denotes the output of $\mathcal{Z}$ after the respective executions.

[6] 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.

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

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 1. *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,*

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

The SUC composition theorem is essentially the same as the UC composition theorem: secure protocols "behave like" ideal functionalities when run in arbitrary environments. See the full version for a formal statement of the theorem.

3 An 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 $\mathcal{F}_{\text{COM}}$, due to its relative simplicity. We also consider realizing the $\mathcal{F}_{\text{ZK}}$ functionality in the $\mathcal{F}_{\text{COM}}$-hybrid model, since existing protocols "gloss over" the details of using the composition theorem correctly.

3.1 Differences in Defining 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]: "*we say that an ideal functionality $\mathcal{F}$ sends a* 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 id), $\mathcal{F}$ 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 1 (Functionality $\mathcal{F}_{\text{COM}}$ 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 Corrupted value to C, and send x to the adversary. Furthermore, if the adversary now provides a value x', and the Receipt output was not yet written on R's tape, then change the recorded value to x'.

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

In contrast, in the SUC framework the functionality description is far simpler. Before writing the functionality, we introduce a convention that was used in [12] for the public headers and private contents in functionalities. The "operation labels" (e.g., Commit, 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. In addition, we parameterize the functionality by some $m = \text{poly}(n)$, which means that all commitment values are of length m. This is needed since SUC parties have a fixed polynomial running time, and so a receiver who does not receive input to the commitment functionality cannot process arbitrarily long strings. Note that all known UC commitment schemes work in this way (i.e., they are either commitments to bits, fixed-length strings, or group elements, etc.). Thus, this definition matches existing constructions.[8] We have:

FIGURE 2 (Functionality $\mathcal{F}_{\text{COM}}$ for the SUC Framework)

$\mathcal{F}_{\text{COM}}$ runs with length parameter m, as follows:

1. Upon receiving an input $(\mathsf{Commit}, sid, x)$ from C, verify that $x \in \{0,1\}^m$ and 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 $\mathcal{F}_{\text{COM}}$

[8] We remark that it is also possible to define $\mathcal{F}_{\text{COM}}$ so that S inputs x and R inputs $1^{|x|}$. This ensures that R can run in time that is polynomial in the length of the committed value. We chose the formulation of a fixed m since it more closely models how UC commitments are typically constructed.

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.

3.2 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 [10, 12], and see [14] for a more recent work following the same paradigm. The authors of [14] claim security of their zero knowledge protocol by referring to the proof of security of zero knowledge from commitments that appears in [10]. 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 [14].

Before proceeding, denote the commitment protocol of [14] by Π_{COM}, the CRS functionality by $\mathcal{F}_{\mathrm{CRS}}$, and the zero knowledge protocol of [10, 14] by Π_{ZK}. Protocol Π_{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 $\mathcal{F}_{\mathrm{MCOM}}$ is used but for simplicity we will ignore this here. Note that the commitment protocol Π_{COM} in [14] uses a fully-homomorphic encryption scheme denoted $\mathrm{Q}_{\mathrm{ENC}}$ and a CCA-secure encryption scheme $\mathrm{ENC}_{\mathrm{CCA}}$.

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 Π_{COM} in [14] is polynomial time is immediate, and simply follows from the fact that the $\mathrm{Q}_{\mathrm{ENC}}$ and $\mathrm{ENC}_{\mathrm{CCA}}$ 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 Π_{COM} in [14] is polynomial time in the full UC framework, one needs to first pad the input of each party in Π_{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 $\mathcal{F}_{\mathrm{CRS}}$. If $\mathcal{F}_{\mathrm{CRS}}$ is assumed to be a local functionality (e.g., secure setup), then this is not difficult since the only input to $\mathcal{F}_{\mathrm{CRS}}$ is the pair (CRS, sid). However, if $\mathcal{F}_{\mathrm{CRS}}$ is implemented via coin-tossing using a local $\mathcal{F}_{\mathrm{CRS}}$ functionality (as suggested in the JUC [13] solution to achieving independent CRS invocations per protocol), then the number of tokens needed to be provided is different. Essentially, a different $\mathcal{F}_{\mathrm{CRS}}$ 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 Π_{ZK} using $\mathcal{F}_{\mathrm{COM}}$. These zero-knowledge protocols make multiple calls to the commitment functionality. The number of calls to $\mathcal{F}_{\mathrm{COM}}$, and thus the length of the input written by the parties in Π_{ZK} to $\mathcal{F}_{\mathrm{COM}}$, 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 $\mathcal{F}_{\mathrm{COM}}$ 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 $\mathcal{F}_{\mathrm{COM}}$, 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 $\mathcal{F}_{\mathrm{COM}}$ can actually depend on the implementation of $\mathcal{F}_{\mathrm{ZK}}$ as used by Π_{COM}. To make this even more complex, if $\mathcal{F}_{\mathrm{COM}}$ uses $\mathcal{F}_{\mathrm{CRS}}$ as described above, then the number of token further depends on whether $\mathcal{F}_{\mathrm{CRS}}$ 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.

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, A. Cohen and Y. Lindell. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation (full version). *Cryptology ePrint Archive*, Report 2014/553, 2014.
10. R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO 2001*, Springer (LNCS 2139), pages 19–40, 2001.
11. R. Canetti and A. Herzberg. Maintaining Security in the Presence of Transient Faults. In *CRYPTO 1994*, Springer (LNCS 839), pages 425–438, 1994.
12. 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.
13. R. Canetti and T. Rabin. Universal Composition with Joint State. In *CRYPTO 2003*, Springer-Verlag (LNCS 2729), pages 265–281, 2003.
14. I. Damgård, A. Polychroniadou and V. Rao. Adaptively Secure UC Constant Round Multi-Party Computation. *Cryptology ePrint Archive*, Report 2014/830, 2014.
15. O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications.* Cambridge University Press, 2004.

16. 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.
17. 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.
18. D. Hofheinz, J. Müller-Quade and D. Unruh. Polynomial Runtime and Composability. *IACR Cryptology ePrint Archive*, report 2009/23, 2009.
19. D. Hofheinz and V. Shoup. GNUC: A New Universal Composability Framework. *IACR Cryptology ePrint Archive*, report 2011/303, 2011.
20. J. Katz, U. Maurer, B. Tackmann, V. Zikas. Universally Composable Synchronous Computation. *Theory of Cryptology Conference (TCC)* 2013: 477-498.
21. R. Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. *CSFW 2006,* pp. 309-320.
22. R. Küsters and M. Tuengerthal. The IITM Model: a Simple and Expressive Model for Universal Composability. *IACR Cryptology ePrint Archive*, report 2013/25, 2013.
23. 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.
24. 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.
25. S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 392–404, 1991.
26. D. Micciancio and S. Tessaro. An Equational Approach to Secure Multi-Party Computation. In *ITCS 2013*, pages 355–372, 2013.
27. 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.
28. D. Wikström. *On the Security of Mix-Nets and Hierarchical Group Signatures.* PhD Thesis, 2005.