A Framework for Universally Composable Publicly Verifiable Cryptographic Protocols

Carsten Baum[1]*, Bernardo David[2]**, and Rafael Dowsley[3]*

[1] Aarhus University
[2] IT University Copenhagen
[3] Bar-Ilan University

Abstract. The Universal Composability (UC) framework (FOCS '01) is the current gold standard for proving security of interactive cryptographic protocols. Proving security of a protocol in UC is an assurance that the theoretical model of a protocol does not have any obvious bugs, in particular when using it as part of a larger construction. UC allows to reason about complex structures in a bottom-up fashion by talking about the individual components and how they are composed. It thereby simplifies the construction of complex secure protocols. Due to certain design choices of the UC framework, realizing certain security notions such as verifiability is cumbersome and "obviously secure" constructions require rather strong and thus in practice expensive individual building blocks. In this work we give the first formal study of Non-Interactive Public Verifiability of UC protocols. As Non-Interactive Public Verifiability is crucial when composing protocols with a distributed ledger, it can be beneficial when designing these with formal security in mind. We give a thorough discussion and formalization of what Non-interactive Public Verifiability means in the Universal Composability Framework and construct a general transformation that achieves this notion for a large class of cryptographic protocols. Our framework furthermore allows to reason about the composition of Non-Interactive Publicly Verifiable primitives.

1 Introduction

Universal Composability (UC) [9] is currently the most popular framework for designing, formalizing and proving security of cryptographic constructions under arbitrary composition. It allows one to capture the interaction between individual parties, their passing of messages and which impact an adversary could have on any protocol in a complex environment consisting of multiple nested protocols. UC, albeit being very general, has seen many extensions such as e.g. UC with joint state [15] or Global UC [11]. On a very high level, UC formalizes

* Supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

** This work was supported by a grant from Concordium Foundation and by DFF grant number 9040-00399B (TrA^{2}C).

cryptographic tasks as ideal functionalities $\mathcal{F}$ with which parties $\mathcal{P}_i$ can interact. An implementation $\mathit{\Pi}$ of such a functionality $\mathcal{F}$ is then considered secure if any interaction of dishonest parties in the real protocol $\mathit{\Pi}$ can be simulated by a probabilistic polynomial time interactive Turing machine (PPT iTM), called the simulator $\mathcal{S}$, which itself may only interact with $\mathcal{F}$ for this purpose. Both the functionality and the parties as well as the adversary $\mathcal{A}$ that controls the dishonest parties, and the external distinguisher (called "the environment" $\mathcal{Z}$) which formalizes the actual security experiment are modeled as PPT iTMs which interact via "interactive" tapes that are shared among these.

The benefit of UC is that, as a formal framework, it allows to discuss the different aspects of an interactive protocol with mathematical precision. But in practice, one often sees that security is argued on a very high level only. This is partially due to the complexity of fully formalizing a protocol in UC, but also stems from the fact that for some, seemingly simple and intuitive cases, the standard UC framework must first be extended to be expressive and applicable.

As an example, consider the class of protocols which interact with a distributed ledger. Albeit Badetscher et al. [3] (based on previous work of Kiayas et al. [21]) have described how to formalize certain distributed ledgers in UC, many important components used in protocols that leverage distributed ledgers still have not been satisfactorily captured in the UC framework, among them functionalities with public verifiability. A popular task in this realm is for example to couple a smart contract with a cryptographic protocol. In such a setting, one must reason about how the distributed ledger can communicate with the functionalities representing a protocol even though the distributed ledger may not have been a part of the actual protocol session. Also the smart contract may have to communicate with such a functionality after the protocol finishes (see e.g. [4] as an example for how complicated this is to achieve). The problem in the aforementioned motivational example seems to be that it is difficult to formalize Non-Interactivity in UC in a meaningful and composable way. A further problem is allowing for third parties who do not participate in the execution to verify its outputs, which gives rise to important issues that will now discuss.

1.1 The Problems of Achieving Verifiability in UC

We will now give an example for why achieving verification in UC is difficult or requires heavy machinery that seems unnecessary from an intuitive point of view and purely like an artifact of UC itself. The notion of verifiability considered in this paper is the following: after the protocol execution, the participants can decide to reveal some of their inputs or outputs to the world. It should be possible for verifiers (even external ones) to check if the revealed values are the ones corresponding to the protocol execution. This is an important notion for applications based on distributed ledgers as discussed above. In particular, many protocols based on distributed ledgers require parties to check the output of a protocol execution on the ledger (*i.e.* such that at least some protocol messages are registered on the ledger) after the execution is completed and/or when the parties involved are offline [18,19,4,21,2]. These protocols essentially require our notion of non-interactive public verifiability but use hardcoded concrete specific mechanisms that achieve this notion instead of generic building blocks. The current lack of a UC treatment of public verifiability is a hurdle that prevents such protocols from being designed and proven in a (even more) modular fashion.

In order to motivate the challenges we must overcome, we will discuss the main issues of capturing public verifiability in the UC model. Let us assume a UC functionality $\mathcal{F}$ which has one round of inputs by the parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, computes some outputs based on the inputs and in the end sends these outputs to each party. Assume that we want to add verifiability to $\mathcal{F}$, meaning that it shall also have interfaces to react to verification requests. Let us call this extended functionality $\mathcal{F}^{\mathtt{V}}$. These requests will be able to confirm if certain inputs or outputs were provided by a party $\mathcal{P}_i$ to the functionality or sent from $\mathcal{F}^{\mathtt{V}}$ to $\mathcal{P}_i$. To achieve this, we design a protocol $\mathit{\Pi}^{\mathtt{V}}$ based on the original UC-secure protocol $\mathit{\Pi}$ that would implement these interfaces. A straightforward solution is to construct $\mathit{\Pi}^{\mathtt{V}}$ in such a fashion that each party would commit to inputs and randomness that it will use in the protocol $\mathit{\Pi}$. The parties then run $\mathit{\Pi}$ based on these committed values. Intuitively, this approach should have a simple verification procedure: each involved party can simply inspect the committed inputs and randomness of other parties, re-run these parties in its head and compare its messages to the available protocol transcript. A third party could then do the same, based on the commitments and a transcript of $\mathit{\Pi}$. Unfortunately, using this approach together with the actual inputs means that we run into adaptivity problems when trying to prove $\mathit{\Pi}^{\mathtt{V}}$ secure. This is similar to [20]: In the security proof of $\mathit{\Pi}^{\mathtt{V}}$, without knowing what the inputs and randomness of any honest party are (as these will only be released by $\mathcal{F}^{\mathtt{V}}$ after the computation has finished), we have to simulate an instance of $\mathit{\Pi}$ with all its messages towards the adversary $\mathcal{A}$. After the simulation finishes, the verification interface of the ideal functionality $\mathcal{F}^{\mathtt{V}}$ would now output the inputs of the honest parties to the simulator. But in the simulation where we ran $\mathit{\Pi}$ with the dishonest parties we had to simulate all the protocol messages assuming dummy inputs for the honest parties. This means that for verification of the instance of $\mathit{\Pi}$ one has to "explain" all those messages that the adversary $\mathcal{A}$ obtained as being possible given the actual inputs of the honest parties from $\mathcal{F}^{\mathtt{V}}$. Towards explaining this, the simulator can essentially only change the randomness of every simulated honest party in the end of the simulation. That essentially implies that $\mathit{\Pi}$ has to be adaptively secure to begin with if we want to prove that $\mathit{\Pi}^{\mathtt{V}}$ UC-securely implements $\mathcal{F}^{\mathtt{V}}$. This means that the implementations of $\mathcal{F}^{\mathtt{V}}$ in practice will be slow, since the set of protocols $\mathit{\Pi}$ which are adaptive is limited. This seems counterintuitive: beyond the technical reason to allow UC simulation, there seems to be no explanation why only adaptive protocols should be verifiable in the aforementioned way.

1.2 Our Approach

In this work, we give a framework that allows to describe non-interactive verifiability in UC and shows how such protocols can be composed. Our framework in its current form is applicable to any functionality that has one round of input as well as multiple rounds of computation and outputs, and captures therefore such functionalities as Oblivious Transfer or Secure Function Evaluation.[4] We furthermore describe a standard wrapper for any such functionality to equip it with the interfaces necessary for non-interactive verification, allowing external verifiers to register (in a public or private manner) and to perform tasks related to verification. This wrapper is particularly designed to amalgamate the reactive nature of UC with non-interactivity.

Our overall approach towards compilation of protocols into verifiable counterparts is a black-box one: we will let one or more parties sacrifice its input and randomness towards establishing correctness of their outputs, as was sketched already in Section 1.1. Such black-box verification procedure also needs access to a transcript. Thus, towards verifiability by third parties, we allow different degrees of assurance about the transcript of the protocol: in addition to committing to inputs and randomness, the compiler will ask parties to either sign their sent or received messages or alternatively authenticate them using what we call "Authentication Functionalities". These Authentication Functionalities do not just have unforgeability properties for those parts of the transcript that are related to honest parties, but also commit dishonest parties to their transcript (without revealing private messages ahead of time or implying consensus on its own). We show how to practically implement these Authentication Functionalities.

As outlined above, the exact use of a certain compiler only fixes some parts of the transcripts. We therefore also formalize the requirements that exchanging the non-committed part of the transcript adaptively cannot break verification as an additional property, which we call "Collusion Resistance". Collusion Resistance can be seen as being complimentary to how much additional apparatus we introduce in the compilation: the more messages an adversary can exchange in a transcript without breaking the verifiability invariant i.e. the stronger the collusion resistance of Π, the weaker are the necessary tools which the compiler has to use in order to achieve verifiability.

Even with the aforementioned framework in place we still have to address the problem of adaptivity which stems from the way how verification is defined. We therefore now sketch the high-level intuition behind our solution which avoids adaptivity (even though the actual formalization differs substantially):

Solution #1: Black-box access to the next messages. Clearly, defining verifiability by just extending $\mathcal{F}$ with some verification commands will not be enough with the aforementioned compilation technique. A first idea is to let the simulation of Π be done inside $\mathcal{F}^{\mathtt{V}}$. This means that we give $\mathcal{F}^{\mathtt{V}}$ an extra "next message function" interface **nmf**. The simulator of $\Pi^{\mathtt{V}}$ will be able to use **nmf** in order to generate the protocol messages in the simulation. We can then easily explain Π consistently after the fact. $\mathcal{F}^{\mathtt{V}}$ can provide **nmf** as it is able to simulate an instance of Π internally using the actual inputs that $\mathcal{F}^{\mathtt{V}}$ obtained from the honest parties. This "internal" simulator will do the input-related work, while the "external" simulator of $\Pi^{\mathtt{V}}$ will handle all interactions

[4] It would be interesting future work to extend our techniques to reactive computations.

with the adversary and functionalities in the actual proof as well as simulation of the compilation overhead. Based on the UC security of Π, making **nmf** available should intuitively not break existing constructions. This is because before we open any inputs or randomness, it should be indistinguishable if we use the actual inputs running Π or use $\mathcal{S}$ due to the UC-security of Π. The "external" simulator for Π^{V} in the security proof then extracts the protocol messages in Π that the adversary sends (and those sent to the hybrid functionalities) and input these accordingly into $\mathcal{F}^{\mathsf{V}}$, *thus generating a valid transcript of* Π. This leads to a new problem: we need to have "internal" simulators that can use real inputs of honest parties *while* simultaneously being able to extract the inputs of the dishonest parties *and* creating an indistinguishable transcript. This extraction requirement stems from the fact that the "external" simulator for Π^{V} may not know the semantics of the original security proof of Π and can therefore not extract the inputs from $\mathcal{A}$.

Solution #2: Extracting inputs. We solve this problem by requiring that for our compiled protocols Π there exists a special "über" simulator which runs something indistinguishable from the actual protocol based on the real inputs (and randomness) of honest parties but is still able to extract inputs of the dishonest parties. Then the "external" simulator of Π^{V} is freed from having to extract the inputs. While requiring that there exists such an über simulator might seem artificial and strong, it actually differs from requiring adaptivity of Π. On one hand, many MPC protocols such as [24,17] simulate their online phase anyway using "artificial" fixed inputs and otherwise run the protocol honestly while still being able to extract. We can therefore directly make those protocols verifiable. At the same time, requiring that the simulation done by the über simulator generates all messages in this consistent way means that we can compile garbled circuits (GC)-based protocols such as the NISC protocol of [13] too. In GC protocols, simulation of such a garbler is trivially possible by running the actual protocol (not having to "simulate" fake GCs) while the simulator can extract inputs of the evaluator from hybrid functionalities such as OT.

Why not using the original UC simulator? As we start out by assuming a UC-secure protocol Π it might be tempting to ask why it is not sufficient to adjust the simulator $\mathcal{S}$ of the original security proof of Π by simply requiring that its inputs can be programmed. For example, in the case of MPC protocols with information-theoretic online phases basically all known UC security proofs actually run a protocol instance on random (or fixed) inputs and then equivocate the protocol outputs to the values obtained from $\mathcal{F}$. As mentioned above, this strategy breaks down when Π is a protocol using Garbled Circuits. In such cases, equivocation is usually achieved by programming the output of $\mathcal{F}$ *into the circuit*. In such a case, revealing the randomness of the garbler immediately shows that a simulation was done. We will nevertheless show that for a certain class of simulators of UC-secure protocols it is possible to construct an über simulators directly.

A High-level Overview of Our Approach: We start with an observation similar to [20], namely that by fixing the inputs, randomness and messages in a protocol we can get a guarantee about the outputs. This is because the inputs and randomness essentially fix both the messages and outputs of each party, as `nmes` is deterministic given all other such values. Therefore our main idea is to authenticate a set of messages as well as inputs and randomness, such that an external party can verify such authenticated items after the fact. On the other hand, fixing all messages that are exchanged in a protocol (in order to leverage this guarantee) is costly, so we generalize this concept to the notion of *collusion resistance* in Section 3.1. Here, we might not have all exchanged messages fixed, but an adversary might be allowed to replace all those messages exchanged between dishonest parties after the fact, or those that were exchanged with the honest parties. Even in such settings, we do not want that the adversary is able to replace the transcript with a new version that is still consistent but lets the parties output a different value. While it might seem obvious that such a protocol with all of its messages fixed would be publicly verifiable, it is still not directly possible to prove this fact in the UC framework unless we assume that Π is adaptively secure - which seems an overkill, given that this is even necessary for verifiable protocols that are statically secure only. In Section 3.2 we therefore introduce *input-aware simulators* (or über simulators) whose transcripts are generated with the actual inputs of the honest parties (and which also extract the inputs of dishonest parties), but whose transcripts should still be indistinguishable from the actual simulation of Π.

1.3 Related Work

Verifiability for several kinds of protocols has been approached from different perspectives, such as cheater identification [20], incoercible secure computation [1], secure computation on public ledgers [2,5,21,22], and improved definitions for widely used primitives [8,7].

1.4 Paper Outline

We start with some preliminaries in Section 2. In Section 3, we present our approach for how to make protocols non-interactively verifiable. In Section 4, we define authentication functionalities that serve as building blocks for our compiler. Our functionalities allow for a set of parties to jointly authenticate messages but do not deliver any message. Finally, we show how to compile protocols into non-interactively verifiable counterparts in Section 5.

2 Preliminaries

We denote the security parameter by κ. We denote concatenation of two strings a and b by $a||b$. Let $y \overset{\$}{\leftarrow} F(x)$ denote running the randomized algorithm F with input x and random coins, and obtaining the output y. When the coins r are specified we use $y \leftarrow F(x;r)$. Similarly, $y \leftarrow F(x)$ is used for a deterministic algorithm. For a set $\mathcal{X}$, let $x \overset{\$}{\leftarrow} \mathcal{X}$ denote x chosen uniformly at random from $\mathcal{X}$; and for a distribution $\mathcal{Y}$, let $y \xleftarrow{\$} \mathcal{Y}$ denote y sampled according to the distribution $\mathcal{Y}$. We will denote by $\mathsf{negl}(\kappa)$ the set of negligible functions of κ. We abbreviate *probabilistic polynomial time* as PPT. Two ensembles $X = \{X_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ and $Y = \{Y_{\kappa,z}\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$ of binary random variables are said to be *statistically indistinguishable*, denoted by $X \approx_s Y$, if for all z it holds that $\mid \Pr[\mathcal{D}(X_{\kappa,z}) = 1] - \Pr[\mathcal{D}(Y_{\kappa,z}) = 1] \mid$ is negligible in κ for every probabilistic distinguisher $\mathcal{D}$. In case this only holds for non-uniform PPT distinguishers we say that X and Y are *computationally indistinguishable* and denote it by $X \approx_c Y$.

2.1 Secure Protocols

A protocol Π consists of the algorithms $\mathtt{nmes}, \mathtt{out}$ and additional parameters: the number of parties n, the resources $\mathcal{F}_1, \ldots, \mathcal{F}_r$, the number of output rounds G, the number of rounds H_τ to obtain each output $\tau \in [G]$ as well as the communication and output model. We assume that some external information s is fixed for the protocol. In an MPC scheme, this information e.g. could consist of the circuit. Each party $\mathcal{P}_i$ uses their respective input x_i as well as randomness r_i for the actual protocol. Here they perform H_τ calls to a next-message function with a subsequent message exchange with both the parties and the resources, finalized by the computation of the τ-th output of the protocol. Formally, the algorithms which comprise Π are as follows:

$\mathtt{nmes}$ is a deterministic polynomial-time (DPT) algorithm which on input the party number i, the protocol input x_i, randomness r_i, auxiliary input s, output round $\tau \in [G]$, current round number $\rho \in [H_\tau]$ and sets of previous messages $\mathcal{M}_{\cdot,i}$ from parties and $\mathcal{N}_{\cdot,i}$ from resources outputs $\{\mathtt{m}_{i,j}^{(\tau,\rho)}\}_{j\in[n]\setminus\{i\}}$, $\{\mathtt{mres}_{i,q}^{(\tau,\rho)}\}_{q\in[r]}$.

$\mathtt{out}$ is a DPT algorithm which on input the party number i, the protocol input x_i, randomness r_i, auxiliary input s as well as output round $\tau \in [G]$, a set of messages $\mathcal{M}_{\cdot,i}$ from parties and $\mathcal{N}_{\cdot,i}$ from resources outputs $\mathtt{y}_i^{(\tau)}$ which is either an output value or $\perp$.

$\mathtt{nmes}$ generates two different types of messages, namely $\mathtt{m}$- and $\mathtt{mres}$-messages. As we shall see later, the $\mathtt{m}$-messages are used for communication *among parties* whereas $\mathtt{mres}$-messages are exchanged *between a party and a functionality*. Therefore, each $\mathtt{mres}$-message consists of an interface ($\mathbf{Input}_i, \mathbf{Compute}^{(\tau)}$, $\mathbf{Output}_i^{(\tau)}$) with whom the party wants to communicate as well as the actual payload. Each message that is an output of $\mathtt{nmes}$ may either be an actual string or a symbol $\perp$, meaning that no message is sent to a certain party/functionality whatsoever in a certain round. For notational consistency, whenever we write $\mathtt{m}_{i,j}$ we mean that a message was sent from party $\mathcal{P}_i$ to $\mathcal{P}_j$. Similarly, we write $\mathtt{mres}_{i,q}$ when the message was sent from $\mathcal{P}_i$ to $\mathcal{F}_q$ and $\mathtt{mres}_{q,i}$ when sent from $\mathcal{F}_q$ to $\mathcal{P}_i$. We will denote messages received by party $\mathcal{P}_i$ from another party as $\mathcal{M}_{\cdot,i}$ and those sent by $\mathcal{P}_i$ to another party as $\mathcal{M}_{i,\cdot}$. Similarly, we will write $\mathcal{N}_{\cdot,i}$ for all messages that $\mathcal{P}_i$ received from resources while $\mathcal{N}_{i,\cdot}$ denotes messages which

$\mathcal{P}_i$ sent to resources. In Figure 1 we describe the general pattern according to which the above algorithms are used in the protocol Π.

Protocol Π

Each $\mathcal{P}_i$ has input x_i as well as common public input s.

Input$_i$: Party $\mathcal{P}_i$ samples r_i uniformly at random. Let $\mathcal{M}_{\cdot,i}, \mathcal{N}_{\cdot,i} \leftarrow \emptyset$.

Compute$^{(\tau)}$: Let $\tau \in [G]$. Then each party $\mathcal{P}_i$ for $\rho \in [H_\tau]$ does the following:

1. Locally compute
$$\left(\{\mathtt{m}_{i,j}^{(\tau,\rho)}\}_{j\in[n]\setminus\{i\}}, \{\mathtt{mres}_{i,q}^{(\tau,\rho)}\}_{q\in[r]}\right) \leftarrow \mathtt{nmes}(i, x_i, r_i, s, \tau, \rho, \mathcal{M}_{\cdot,i}, \mathcal{N}_{\cdot,i}).$$
2. For each $j \in [n] \setminus \{i\}$ send $\mathtt{m}_{i,j}^{(\tau,\rho)}$ to $\mathcal{P}_j$. Furthermore, for each $q \in [r]$ send $\mathtt{m}_{i,q}^{(\tau,\rho)}$ to $\mathcal{F}_q$.
3. For each $j \in [n] \setminus \{i\}$ wait for $\mathtt{m}_{j,i}^{(\tau,\rho)}$ from each $\mathcal{P}_j$ as well as $\mathtt{mres}_{q,i}^{(\tau,\rho)}$ from each $\mathcal{F}_q$ for $q \in [r]$.
4. Set $\mathcal{M}_{\cdot,i} \leftarrow \mathcal{M}_{\cdot,i} \cup \{\mathtt{m}_{j,i}^{(\tau,\rho)}\}_{j\in[n]\setminus\{i\}}$ and $\mathcal{N}_{\cdot,i} \leftarrow \mathcal{N}_{\cdot,i} \cup \{\mathtt{mres}_{q,i}^{(\tau,\rho)}\}_{q\in[r]}$.

Output$_i^{(\tau)}$: Party $\mathcal{P}_i$ locally computes and outputs
$$\mathtt{y}_i^{(\tau)} \leftarrow \mathtt{out}(i, x_i, r_i, s, \tau, \mathcal{M}_{\cdot,i}, \mathcal{N}_{\cdot,i}).$$
given $\tau \in [G]$.

Fig. 1. The generic protocol Π.

Communication Model. Generally, we do not make any restriction on the messages that are exchanged (except that their length is polynomial in the security parameter). These will be sent through point-to-point secure channels by them. We call this setting *private communication.* If the parties instead send the same message to all other parties, then we consider this as *broadcast communication.* Parties may arbitrarily mix private and broadcast communication. We require that all message-passing is synchronous.

Output Model. As with communication, the above definition does not restrict the output $\mathtt{y}_i^{(\tau)}$ which each party obtains in the end of the computation. This general setting where all the $\mathtt{y}_i^{(\tau)}$ might be completely different is what we consider as *asymmetric output.* This is the standard for many interesting functions that one can compute, such as e.g. Oblivious Transfer. If all parties compute the same output then we call this *symmetric output.*

2.2 Universal Composition of Secure Protocols

In this work we use the (Global) Universal Composability or (G)UC model [9,11] for analyzing security and refer interested readers to the original works for more details. Naturally, we only discuss the dishonest-majority setting in this work as honest-majority protocols can simply output a vote of all parties if the result is correct or not (if broadcast is available).

In this work we consider protocols that are run by interactive Turing Machines (iTMs) which we call *parties*. A protocol Π will have n parties which we denote as $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$. We assume that each party runs in probabilistic polynomial time (PPT) in some implicit security parameter κ. The *adversary* $\mathcal{A}$, which also is a PPT iTM, will be able to corrupt parties, but we only allow him to corrupt up to a threshold of $k < n$ of them, though non-threshold adversary structures may also be supported. We opt for the static corruption model where the parties are corrupted from the beginning, as this is what most efficient protocols currently are developed for. The set of corrupted parties is denoted as $I \subset \mathcal{P}$. The parties will be able to exchange messages with each other and also with resources, which we call (as is standard in the literature) *ideal functionalities* (which themselves are PPT iTMs). To simplify notation we do assume that the messages between parties are sent over secure channels, as well as their communication with the ideal functionalities.

We start out with protocols that are themselves already secure, but not verifiable. For this, we assume that the ideal functionality $\mathcal{F}$ of a protocol Π follows the pattern as described in Figure 2. In there, we consider protocols where parties give input initially, but obtain possibly G rounds of output. Having multiple rounds of outputs can be seen as a trade-off: on one hand, it allows us to model e.g. commitment schemes which would not be possible having only one round of output. At the same time, it is not general enough to permit reactive computations which inherently make the notation a lot more complex.

It is not necessary that all of the interfaces which $\mathcal{F}$ provides are used for an application. For example in the case of coin tossing, no party $\mathcal{P}_i$ ever has to call $\mathbf{Input}_i$. While $\mathbf{Input}_i, \mathbf{Output}_i^{(\tau)}$ are fixed in their semantics, the application may freely vary how $\mathbf{Compute}^{(\tau)}$ may act upon the inputs or generate outputs. The only constraint that we make is that for each of the $\tau \in [G]$ rounds, $\mathbf{Compute}^{(\tau)}$ sets output values $(\mathtt{y}_1^{(\tau)}, \ldots, \mathtt{y}_n^{(\tau)})$.

As usual, we define security with respect to a PPT iTM $\mathcal{Z}$ called *environment*. The environment provides inputs to and receives outputs from the parties $\mathcal{P}$. Furthermore, the adversary $\mathcal{A}$ will corrupt parties $I \subset \mathcal{P}$ in the name of $\mathcal{Z}$ and thus gain control over these parties, i.e. will see and be able to generate the protocol messages. To define security, let $\Pi \circ \mathcal{A}$ be the distribution of the output of an arbitrary $\mathcal{Z}$ when interacting with $\mathcal{A}$ in a real protocol instance Π. Furthermore, let $\mathcal{S}$ denote an *ideal world adversary* and $\mathcal{F} \circ \mathcal{S}$ be the distribution of the output of $\mathcal{Z}$ when interacting with parties which run with $\mathcal{F}$ instead of Π and where $\mathcal{S}$ takes care of adversarial behavior.

Functionality $\mathcal{F}$

The functionality works for n parties and has common public input s. Upon initialization, $\mathcal{S}$ is allowed to corrupt a set $I \subset [n]$ of parties where $|I| \leq k$ and $k < n$. Albeit having many interfaces, they fall into 3 different categories for providing inputs as well as running the G evaluation and output steps.

Input$_i$: On input $(\textsc{Input}, sid, x_i)$ by $\mathcal{P}_i$ and $(\textsc{Input}, sid)$ by all other parties store x_i locally and send $(\textsc{Input}, sid, i)$ to all parties. Every further message to this interface is discarded and once set, x_i may not be altered anymore.

Compute$^{(\tau)}$: On input $(\textsc{Compute}, sid, \tau)$ by a set of parties $J_\tau \subseteq [n]$ as well as $\mathcal{S}$ perform a computation based on s as well as the current state of the functionality. The computation is to be specified in concrete implementations of this functionality. The last two steps of this interface are fixed and as follows:

1. Set some values $\mathtt{y}_1^{(\tau)}, \cdots, \mathtt{y}_n^{(\tau)}$. Only this interface is allowed to alter $\mathtt{y}_1^{(\tau)}, \cdots, \mathtt{y}_n^{(\tau)}$.
2. Send $(\textsc{Compute}, sid, \tau)$ to every party in J_τ.

Every further call to **Compute**$^{(\tau)}$ is ignored. Every call to this interface before all **Input**$_i$ are finished is ignored, as well as when **Compute**$^{(\tau-1)}$ has not finished yet.

Output$_i^{(\tau)}$: On input $(\textsc{Output}, sid, \tau)$ by $\mathcal{P}_i$ where $\tau \in [G]$ and if $\mathtt{y}_i^{(\tau)}$ was set send $(\textsc{Output}, sid, \tau, \mathtt{y}_i^{(\tau)})$ to $\mathcal{P}_i$.

Fig. 2. The generic functionality $\mathcal{F}$.

Definition 1 (Secure Protocol). *We say that $\mathcal{F}$ securely implements Π if for every PPT iTM $\mathcal{A}$ there exists a PPT iTM $\mathcal{S}$ (with black-box access to $\mathcal{A}$) such that no PPT environment $\mathcal{Z}$ can distinguish $\Pi \circ \mathcal{A}$ from $\mathcal{F} \circ \mathcal{S}$ with non-negligible probability.*

In our protocols we use the standard digital signature functionality $\mathcal{F}_{\mathsf{Sig}}$ from [10], the key registration functionality $\mathcal{F}_{\mathsf{Reg}}$ from [12] and an authenticated bulletin board functionality $\mathcal{F}_{\mathsf{BB}}$, which are described in Appendix A. We also use constructions of IND-CCA public key encryption schemes that UC-realize the standard public key encryption functionality that are described in Appendix B.

2.3 Verifiable Functionalities

We extend the functionality $\mathcal{F}$ from Section 2.2 to provide a notion of non-interactive verification using a functionality wrapper $\mathcal{F}^{\mathsf{V}}$ described in Figure 3. For this, we assume that there are additional parties $\mathcal{V}_i$ which can partake in the verification. These, as well as regular protocol parties, can register at runtime to be verifiers of the computation using a special interface **Register Verifier**. Once they are registered, these verifiers are allowed to check the validity of outputs for parties that have initiated verification at any point. We keep track of this using the set of verifiers $\mathcal{V}$ (which is initially empty) inside the functionality. For values whose output has been provided using the interface **Output**$_i^{(\tau)}$ (that we

Functionality Wrapper $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$

The functionality wrapper $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$ adds the interfaces below to a generic functionality $\mathcal{F}$ defined as in Figure 2, still allowing direct access to $\mathcal{F}$. $\mathcal{F}^{\mathtt{V}}$ maintains binary variables $\mathsf{verification\text{-}active}, \mathsf{verify\text{-}1}, \ldots, \mathsf{verify\text{-}n}$ that are initially 0, which are used to keep track of the verifiable outputs. Apart from the parties defined in $\mathcal{F}$, $\mathcal{F}^{\mathtt{V}}$ interacts with verifiers $\mathcal{V}_i \in \mathcal{V}$.

Register Verifier (private): Upon receiving $(\textsc{Register}, \mathit{sid})$ from $\mathcal{V}_i$:

- If $\mathsf{verification\text{-}active} = 1$ send $(\textsc{Register}, \mathit{sid}, \mathcal{V}_i)$ to $\mathcal{S}$. If $\mathcal{S}$ answers with $(\textsc{Register}, \mathit{sid}, \mathcal{V}_i, \mathit{ok})$, set $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_i$ and return $(\textsc{Registered}, \mathit{sid})$ to $\mathcal{V}_i$.
- If $\mathsf{verification\text{-}active} = 0$ return $(\textsc{Verification-Inactive}, \mathit{sid})$ to $\mathcal{V}_i$.

Register Verifier (public): Upon receiving $(\textsc{Register}, \mathit{sid})$ from $\mathcal{V}_i$:

- If $\mathsf{verification\text{-}active} = 1$ set $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_i$ and return $(\textsc{Registered}, \mathit{sid})$ to $\mathcal{V}_i$.
- If $\mathsf{verification\text{-}active} = 0$ return $(\textsc{Verification-Inactive}, \mathit{sid})$ to $\mathcal{V}_i$.

Activate Verification: Upon receiving $(\textsc{Activate-Verification}, \mathit{sid}, \mathsf{open\text{-}i}, \mathsf{open\text{-}input\text{-}i})$ from each $\mathcal{P}_i$ and if $\mathbf{Compute}^{(1)}, \ldots, \mathbf{Compute}^{(G)}$ succeeded:

1. Let $Y \leftarrow \{i \in [n] \mid \mathsf{open\text{-}i} = 1 \wedge \mathsf{verify\text{-}i} = 0\}$. If $Y = \emptyset$ then return.
2. Set $\mathsf{verification\text{-}active} \leftarrow 1$ (if it is not set already) and deactivate the interfaces $\mathbf{Compute}^{(\tau)}$ for all $\tau \in [G]$.
3. If $\mathsf{open\text{-}input\text{-}i} = 1$, then set $z_i = x_i$; otherwise $z_i = \perp$.
4. Send $(\textsc{Activating-Verification}, \mathit{sid}, Y, \{z_j, \mathtt{y}_j^{(\tau)}\}_{j \in Y, \tau \in [G]})$ to $\mathcal{S}$.
5. Upon receiving $(\textsc{Activating-Verification}, \mathit{sid}, \mathit{ok})$ from $\mathcal{S}$ set $\mathsf{verify\text{-}i} \leftarrow 1$ for each $i \in Y$. Then return $(\textsc{Verification-Activated}, \mathit{sid}, Y, \{z_j, \mathtt{y}_j^{(\tau)}\}_{j \in Y, \tau \in [G]})$ to all parties in $\mathcal{P}$.
6. Upon outputting inputs of an honest party $\mathcal{P}_i$ to $\mathcal{S}$ append $\mathcal{P}_i$'s randomness r_i.

Verify$_j$: Upon receiving $(\textsc{Verify}, \mathit{sid}, j, a, b^{(1)}, \ldots, b^{(G)})$ from $\mathcal{V}_i$ where $\mathcal{V}_i \in \mathcal{V}$ and $\mathcal{P}_j \in \mathcal{P}$ do the following:

- if $\mathsf{verify\text{-}j} = 1$ then compute the set $B \leftarrow \{\tau \in [G] \mid b^{(\tau)} \neq \mathtt{y}_j^{(\tau)}\}$. If $a = z_j$, then set $f \leftarrow a$; otherwise $f \leftarrow \perp$. Return $(\textsc{Verify}, \mathit{sid}, j, f, B)$ to $\mathcal{V}_i$.
- If $\mathsf{verify\text{-}j} = 0$ then send $(\textsc{Cannot-Verify}, \mathit{sid}, j)$ to $\mathcal{V}_i$.

Input$_i$: On input $(\textsc{Input}, \mathit{sid}, r_i, x_i)$ by $\mathcal{P}_i$ and $(\textsc{Input}, \mathit{sid})$ by all other parties: if $i \in \overline{I}$ then forward $(\textsc{Input}, \mathit{sid}, x_i)$ to $\mathcal{F}$ and also forward responses from $\mathcal{F}$ to the original sender. If $i \in I$ then ignore the message. Finally, initialize $\mathcal{S}^{\mathtt{U}}$ parameterizing it with r_i and x_i for each simulated honest party $\mathcal{P}_i, i \in \overline{I}$.

Output$_i^{(\tau)}$: On input $(\textsc{Output}, \mathit{sid}, \tau)$ by $\mathcal{P}_i$ and if $i \in \overline{I}$ then forward the message to $\mathcal{F}$ and also forward responses from $\mathcal{F}$ to the original sender. If $i \in I$ then ignore.

nmf: Upon input $(\textsc{NextMsgP}, \mathit{sid}, j, \tau, \rho, \{\mathtt{m}_{i,j}\}_{i \in I})$ where $j \in \overline{I}$ or $(\textsc{NextMsgF}, \mathit{sid}, q, \tau, \rho, \mathtt{mres}_{i,q})$ where $i \in I$ and $q \in [r]$ by $\mathcal{S}$ send the respective message to $\mathcal{S}^{\mathtt{U}}$ and forward its response $(\textsc{NextMsgP}, \mathit{sid}, j, \tau, \rho + 1, \{\mathtt{m}_{j,i}\}_{i \in I})$ or $(\textsc{NextMsgF}, \mathit{sid}, q, \tau, \rho + 1, \mathtt{mres}_{q,i})$ to $\mathcal{S}$.

Fig. 3. The Functionality wrapper $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. Interfaces **Input**$_i$, **Output**$_i^{(\tau)}$ and **nmf** are discussed in Section 3.2

inherit from the definition of $\mathcal{F}$ of Section 2.2) we allow the parties $\mathcal{P}$ to use an interface called **Activate Verification** to enable everyone in $\mathcal{V}$ to check their outputs via the interface **Verify**$_i$. Interfaces **Input**$_i$, **Output**$_i^{(\tau)}$ and **nmf** are related to the notion of über simulators discussed in Section 3.2.

Notice that, in our constructions, a verifier $\mathcal{V}_i \in \mathcal{V}$ can perform verification with help from data obtained in mainly two different ways: 1. receiving verification data from another verifier $\mathcal{V}_j \in \mathcal{V}$ or a party $\mathcal{P}_i \in \mathcal{P}$; 2. retrieving verification data directly from publicly available resource such as a Bulletin Board (represented as a setup functionality). In case $\mathcal{V}_i$ attempts to obtain verification data from another party in $\mathcal{V} \cup \mathcal{P}$, that party might be corrupted, allowing the ideal adversary $\mathcal{S}$ to interfere (*i.e.* providing corrupted verification data or not answering at all). On the other hand, when $\mathcal{V}_i$ obtains such verification data from a resource available as setup (*i.e.* a resource guaranteed to be untamperable by the adversary), $\mathcal{S}$ has no control over the verification process. In order to model the situation where verification data is obtained reliably and that where it is obtained unreliably, $\mathcal{F}^{\mathtt{V}}$ might implement only **Register Verifier (public)** or only **Register Verifier (private)**, respectively. We do not require $\mathcal{F}^{\mathtt{V}}$ to implement both of these interfaces, and thus define the properties of $\mathcal{F}^{\mathtt{V}}$ according to which of them is implemented, according to Definitions 2 and 3.

Definition 2 (Verifier Registration). *Let $\mathcal{F}$ be a functionality which implements the interface* **Register Verifier (public)***, then $\mathcal{F}$ is said to have* Public Verifier Registration*. If $\mathcal{F}$ instead implements* **Register Verifier (private)** *then we say that it has* Private Verifier Registration*.*

Definition 3 (NIV). *Let $\mathcal{F}$ be a functionality which implements the above interfaces* **Activate Verification** *and* **Verify**$_j$ *and which has* Verifier Registration *according to Definition 2, then we call $\mathcal{F}$* NIV*. If $\mathcal{F}$ has* Public Verifier Registration *then $\mathcal{F}$ is* Publicly Verifiable *whereas we call it* Privately Verifiable *if $\mathcal{F}$ has* Private Verifier Registration*.*

3 Verifiability

We now present our approach for making protocols non-interactively verifiable.

3.1 Collusion Resistance of Protocols

We will establish correctness of an output by using the messages of the protocol transcript as well as some properties of $\Pi = (\mathtt{nmes}, \mathtt{out})$ which we will define next. During verification, we want to ensure that $\mathcal{P}_i$ acted honestly during the actual protocol. Unfortunately, if a party $\mathcal{P}_i$ is dishonest then messages which he claims to have sent or received might have been altered by the adversary, in which case we do not want that the verification passes for an output $\widehat{\mathtt{y}_i^{(\tau)}} \notin \{\mathtt{y}_i^{(\tau)}, \perp\}$ of $\mathcal{P}_i$. As an honest party would have never activated verification if the result was $\perp$, this automatically leads to detection of cheating.

We now define the security game necessary for our compiler. In this security game, an adversary $\mathcal{A}$ will adaptively generate two protocol transcripts together with honest parties whose inputs and randomness it does not know initially. The adversary will output both transcripts which must coincide according to the definition that now follows, and will win if they coincide while the outputs of some party are different and not $\perp$. We want to cover a diverse range of protocols which might come with different levels of guarantees. Therefore, in our definition of the property concerning transcripts which the game implies, called *collusion resistance*, we consider diverse scenarios regarding: (1) whether the parties are committed to their inputs and randomness in the beginning of the execution (parameter ν); (2) what the set of parties CIR is that will reveal their input and randomness later; and (3) which protocol messages the adversary can replace when he attempts to break the verifiability by presenting a fake transcript (parameter μ). While the choices for CIR are evident, we now discuss the different values which ν and μ might have and what this means for the strength of $\mathcal{A}$:

$\nu = \mathsf{ncir}$**:** The parties *are not committed* to the input and randomness in the beginning of the execution.
$\nu = \mathsf{cir}$**:** The parties *are committed* to the input and randomness in the beginning of the execution.
$\mu = \mathsf{ncmes}$**:** $\mathcal{A}$ can replace all messages *by all parties.*
$\mu = \mathsf{chsmes}$**:** $\mathcal{A}$ can replace messages *from corrupted senders.*
$\mu = \mathsf{chmes}$**:** $\mathcal{A}$ can replace messages exchanged *between corrupted parties.*
$\mu = \mathsf{cmes}$**:** $\mathcal{A}$ cannot replace *any message.*

Based on this, we define collusion resistance as follows:

Definition 4. *Let Π be a synchronous protocol that is secure against a static adversary corrupting up to k out of n parties using r resources which are Non-Interactively Verifiable (NIV). For $\nu \in \{\mathsf{cir}, \mathsf{ncir}\}$, $\mu \in \{\mathsf{ncmes}, \mathsf{chsmes}, \mathsf{chmes}, \mathsf{cmes}\}$ and for $\mathsf{CIR} \subseteq [n]$, we define the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$:*

1. *Both $\mathcal{A}, \mathcal{C}$ obtain s as in Π. $\mathcal{C}$ sets up instances of the resources $\mathcal{F}_1, \ldots, \mathcal{F}_r$.*
2. *$\mathcal{A}$ chooses $I \subset [n], |I| \leq k$ and sends I to $\mathcal{C}$. Let $\overline{I} = [n] \setminus I$.*
3. *$\mathcal{C}$ chooses inputs $\{x_i\}_{i \in \overline{I}}$ and samples r_i for $i \in \overline{I}$. If $\nu = \mathsf{cir}$, $\mathcal{A}$ sends $\{x_j, r_j\}_{j \in I}$ to $\mathcal{C}$.*
4. *For all $i \in [n]$, $\mathcal{C}$ sets $\mathcal{M}_{\cdot,i}, \mathcal{M}_{i,\cdot}, \mathcal{N}_{i,\cdot}, \mathcal{N}_{\cdot,i} \leftarrow \emptyset$.*
5. *For each $\tau \in [G]$:*
   (a) *For each $\rho \in [H_\tau]$:*
      i. *For $i \in \overline{I}$, $\mathcal{C}$ computes*

$$\left(\{\mathtt{m}_{i,j}^{(\tau,\rho)}\}_{j \in [n] \setminus \{i\}}, \{\mathtt{mres}_{i,q}^{(\tau,\rho)}\}_{q \in [r]}\right) \leftarrow \mathtt{nmes}(i, x_i, r_i, s, \tau, \rho, \mathcal{M}_{\cdot,i}, \mathcal{N}_{\cdot,i})$$

and then sends $\{\mathtt{m}_{i,j}^{(\tau,\rho)}\}_{i \in \overline{I}, j \in I}$ to $\mathcal{A}$ and $\mathtt{mres}_{i,q}^{(\tau,\rho)}$ to $\mathcal{F}_q$ for $q \in [r]$.

   ii. *Allow $\mathcal{A}$ to interact with all $\mathcal{F}_q$ as defined in the functionalities.*
   iii. *$\mathcal{A}$ sends $\{\mathtt{m}_{j,i}^{(\tau,\rho)}\}_{i\in\overline{I},j\in I}$ to $\mathcal{C}$. If $\mu = \mathsf{cmes}$, $\mathcal{A}$ also sends $\{\mathtt{m}_{j,i}^{(\tau,\rho)}\}_{i\in I,j\in I}$.*
   iv. *Let $\mathtt{mres}_{q,i}^{(\tau,\rho)}$ be the messages that each $\mathcal{F}_q$ sends to parties $\mathcal{P}_i, i \in \overline{I}$.*
   v. *For $i \in [n]$, $\mathcal{C}$ appends[5] $\{\mathtt{m}_{j,i}^{(\tau,\rho)}\}_{j\in[n]\setminus\{i\}}$ to $\mathcal{M}_{\cdot,i}$, $\{\mathtt{mres}_{q,i}^{(\tau,\rho)}\}_{q\in[r]}$ to $\mathcal{N}_{\cdot,i}$, $\{\mathtt{m}_{i,j}^{(\tau,\rho)}\}_{j\in[n]\setminus\{i\}}$ to $\mathcal{M}_{i,\cdot}$ as well as $\{\mathtt{mres}_{i,q}^{(\tau,\rho)}\}_{q\in[r]}$ to $\mathcal{N}_{i,\cdot}$.*

6. *$\mathcal{C}$ for $\{x_i^b, r_i^b, \mathcal{N}_{i,\cdot}^b, \mathcal{N}_{\cdot,i}^b\}_{i\in\overline{I}}^{b\in\{0,1\}}$ and $\{\mathcal{M}_{i,j}^0, \mathcal{M}_{i,j}^1\}_{i,j\in[n],i\neq j}$ does the following*
   (a) *Set $r_i^0, r_i^1 \leftarrow r_i$ and $x_i^0, x_i^1 \leftarrow x_i$ for $i \in \overline{I}$.*
   (b) *Set $\mathcal{N}_{i,\cdot}^0, \mathcal{N}_{i,\cdot}^1 \leftarrow \mathcal{N}_{i,\cdot}$ and $\mathcal{N}_{\cdot,i}^0, \mathcal{N}_{\cdot,i}^1 \leftarrow \mathcal{N}_{\cdot,i}$ for all $i \in \overline{I}$.*
   (c) *Initially set $\mathcal{M}_{i,j}^0, \mathcal{M}_{i,j}^1$ as follows:*
      – *If $\mu = \mathsf{cmes}$, then set $\mathcal{M}_{i,j}^0 = \mathcal{M}_{i,j}^1 \leftarrow \mathcal{M}_{i,j}$ for all $i, j \in [n]$.*
      – *If $\mu = \mathsf{chmes}$, then set $\mathcal{M}_{i,j}^0 = \mathcal{M}_{i,j}^1 \leftarrow \mathcal{M}_{i,j}$ for all $i, j \in [n]$ such that either $i \in \overline{I}$ or $j \in \overline{I}$.*
      – *If $\mu = \mathsf{chsmes}$, then set $\mathcal{M}_{i,j}^0 = \mathcal{M}_{i,j}^1 \leftarrow \mathcal{M}_{i,j}$ for all $j \in [n], i \in \overline{I}$.*
   (d) *Send $\{x_i^b, r_i^b, \mathcal{N}_{i,\cdot}^b, \mathcal{N}_{\cdot,i}^b\}_{i\in\overline{I}}^{b\in\{0,1\}}$ and all defined $\{\mathcal{M}_{i,j}^0, \mathcal{M}_{i,j}^1\}_{i,j\in[n],i\neq j}$ to $\mathcal{A}$.*

7. *For $\{x_j^b, r_j^b, \mathcal{N}_{j,\cdot}^b, \mathcal{N}_{\cdot,j}^b\}_{j\in I}^{b\in\{0,1\}}$ and $\{\mathcal{M}_{i,j}^0, \mathcal{M}_{i,j}^1\}_{i,j\in[n],i\neq j}$ that are undefined, do the following:*
   (a) *If $\nu = \mathsf{ncir}$ then $\mathcal{A}$ chooses $\{r_j^0, r_j^1\}_{j\in I}$, $\{x_j^0, x_j^1\}_{j\in I}$ and sends them to $\mathcal{C}$. If instead $\nu = \mathsf{cir}$, $\mathcal{C}$ sets $r_j^0, r_j^1 \leftarrow r_j$ and $x_j^0, x_j^1 \leftarrow x_j$ for $j \in I \cap \mathsf{CIR}$.*
   (b) *For all $i, j \in [n]$ where $\mathcal{M}_{i,j}^0, \mathcal{M}_{i,j}^1$ were not set in Step 6, $\mathcal{A}$ chooses the values of $\mathcal{M}_{i,j}^0, \mathcal{M}_{i,j}^1$.*
   (c) *For each $j \in I$ let $\mathcal{A}$ provide the sets $\mathcal{N}_{j,\cdot}^0, \mathcal{N}_{j,\cdot}^1, \mathcal{N}_{\cdot,j}^0, \mathcal{N}_{\cdot,j}^1$.*

8. *$\mathcal{C}$ checks the following conditions:*
   – *For each $q \in [r], j \in I$ check that $\mathcal{N}_{\cdot,j}$ are consistent with the messages that $\mathcal{P}_j$ should have obtained from $\mathcal{F}_q$ via the verification interface. If $\mathcal{F}_q$ is furthermore a NIV which allows the verification of inputs from $\mathcal{P}_j, j \in \mathsf{CIR}$, test if $\mathcal{N}_{j,\cdot}$ is consistent with the inputs which $\mathcal{P}_j$ provided to $\mathcal{F}_q$. Otherwise $\mathcal{C}$ outputs 0.*
   – *For each $b \in \{0,1\}, i \in [n], \tau \in [G]$ and $\rho \in [H_\tau]$: if $\mathcal{M}_{i,\cdot}^b, \mathcal{N}_{i,\cdot}^b$ are inconsistent with the output of $\mathtt{nmes}(i, x_i^b, r_i^b, s, \tau, \rho, \mathcal{M}_{\cdot,i}^b, \mathcal{N}_{\cdot,i}^b)$ $\mathcal{C}$ returns 0.*
   – *$\mathcal{C}$ outputs 1 if there exists $t \in [n], \tau \in [G]$ such that*

$$\mathtt{out}(t, x_t^0, r_t^0, s, \tau, \mathcal{M}_{\cdot,t}^0, \mathcal{N}_{\cdot,t}^0) \neq \mathtt{out}(t, x_t^1, r_t^1, s, \tau, \mathcal{M}_{\cdot,t}^1, \mathcal{N}_{\cdot,t}^1)$$

   and both are not $\perp$. Otherwise $\mathcal{C}$ outputs 0.

Then we call a protocol (ν, CIR, μ)-collusion-resistant *if any PPT algorithm $\mathcal{A}$ can make $\mathcal{C}$ output 1 in the above game only with negligible probability.*

[5] Those values which are defined by either being chosen by $\mathcal{C}$ or sent by $\mathcal{A}$.

As Definition 4 is rather complex[6], we want to discuss its repercussions. $\mathcal{C}$ runs the protocol together with $\mathcal{A}$, who *after the protocol adaptively* can alter messages both between parties and parties and functionalities. In particular, $\mathcal{A}$ may first see the choices of the honest parties before announcing his own choices. The parameter μ describes the degree to which $\mathcal{A}$ is allowed to alter messages. As mentioned, ν decides to which extend the inputs and the randomness of each party are fixed - those of the honest parties will always be fixed, but $\mathcal{A}$ may alter them for the dishonest parties in the security experiment. Concerning messages to and from functionalities, $\mathcal{A}$ may always alter them for the dishonest parties but due to the NIV-property of each used functionality this can be detected and consistency can be checked. Furthermore, we also test that all messages which are sent are consistent with the `nmes`-function, which is possible as both inputs and randomness of each party are always revealed.

3.2 Simulating Verifiable Protocols

Most standard simulators $\mathcal{S}$ for UC secure protocols Π work by executing an internal copy of the adversary $\mathcal{A}$ towards which they simulate interactions with simulated honest parties and ideal functionalities in the hybrid model where Π is defined. In general, such a simulator $\mathcal{S}$ receives no external advice and generates random inputs for simulated honest parties and simulated ideal functionality responses with the aid of a random input tape, from which it samples all necessary values. However, a crucial point in our framework is being able to parameterize the operation of certain simulators, as well as giving them external input to how queries to simulated functionalities should be answered.

We need simulators with such properties in order to obtain publicly verifiable versions of existing protocols without requiring them to be adaptively secure as outlined before. Basically, in the publicly verifiable version of a protocol, we wish to embed its original simulator $\mathcal{S}$ in the publicly verifiable functionality that it realizes. This will allow us to "delegate" the simulation of the original protocol to its own simulator, while the simulator for the publicly verifiable version handles only the extra machinery needed to obtain public verifiability. The advantage of this technique is twofold: (1) It allows us to construct publicly verifiable versions of statically secure protocols; (2) It simplifies the security analysis of publicly verifiable versions of existing UC-secure protocols, since only the added machinery for public verifiability must be analysed.

We will now define the notion of an *über simulator* for a UC-secure protocol Π. To distinguish these from the simulator $\mathcal{S}$ which is used in the original UC proof that Π realizes a functionality $\mathcal{F}$ we denote über simulators as $\mathcal{S}^{\mathtt{U}}$. Basically, an über simulator $\mathcal{S}^{\mathtt{U}}$ extends a standard simulator $\mathcal{S}$ in such a way that it is possible to parameterize the simulator with the inputs and randomness to be used by the honest parties that are simulated in interactions with

[6] The length of Definition 4 is mostly due to the fact that it has to handle two full protocol transcripts and shortening it by considering all combinations of ν, μ separately would create 8 almost identical definitions.

the internal copy of the adversary. Moreover, an über simulator allows for responses to queries to simulated functionalities to be given externally and will also output the input and randomness of simulated honest parties. Otherwise $\mathcal{S}^{\mathtt{U}}$ will perform similar actions as a regular simulator, such as extracting inputs of dishonest parties to be sent to $\mathcal{F}$. It is important to remark that most existing simulators for protocols realizing the vast majority of natural UC functionalities already achieve our notion of über simulator or can be easily modified to achieve it. Notice that most simulators basically execute the protocol as an honest party would, except that they use random inputs and later take advantage of their power over setup functionalities to equivocate the result of the protocol such that it yields the actual result obtained by executing with certain inputs (held by honest parties). Departing from such a simulator, an über simulator can be easily constructed by allowing the simulated honest party inputs/randomness to be obtained externally, rather than being randomly generated internally.
Input-aware simulation. Let $\mathcal{S}^{\mathtt{U}}$ be a PPT iTM with the same input and output tapes as the regular simulator $\mathcal{S}$, plus a few additional ones:

Input tapes: Input tapes for the input from the environment $\mathcal{Z}$, individual tapes for randomness and inputs to be used by simulated honest parties, an input tape for randomness r to be used internally, a tape for messages from the adversary $\mathcal{A}$, a tape for messages from the global setup ideal functionalities in the hybrid model where Π is defined and a tape to receive messages from the functionality $\mathcal{F}$.

Output tapes: $\mathcal{S}^{\mathtt{U}}$ furthermore has output tapes to communicate with $\mathcal{F}, \mathcal{Z}, \mathcal{A}$, the ideal functionalities in the hybrid model where Π is defined as well as a special "control output tape" that outputs the input and randomness used by simulated honest parties.

This means that we define $\mathcal{S}^{\mathtt{U}}$ with respect to the ideal functionality $\mathcal{F}$, similar to the normal simulator $\mathcal{S}$. We furthermore define the following two properties of *simulation- and execution-consistency.* Simulation consistency is straightforward and says that any regularly simulated transcript is indistinguishable from a transcript generated by $\mathcal{S}^{\mathtt{U}}$ when operating as $\mathcal{S}$ does, using uniform randomness as well as sampling responses to queries to simulated setup functionalities and simulated party inputs as $\mathcal{S}$ would (without taking external advice).

Definition 5 (Simulation Consistency). *We say that the PPT iTM $\mathcal{S}^{\mathtt{U}}$ is* Simulation-consistent *if no PPT iTM $\mathcal{Z}$ can distinguish the views of*

1. *$\mathcal{F} \circ \mathcal{S}$: An ideal execution of $\mathcal{F}$ and $\mathcal{S}$ executing an internal copy of adversary $\mathcal{A}$ with global setup ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_r$; and*
2. *$\mathcal{F} \circ \mathcal{S}^{\mathtt{U}}$: An ideal execution of $\mathcal{F}$ with $\mathcal{S}^{\mathtt{U}}$ accessing a copy of $\mathcal{A}$ with global setup ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_r$,*

where $\mathcal{S}^{\mathtt{U}}$ operates as $\mathcal{S}$ does: it has direct access to global setup ideal functionalities functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_r$ and to a copy of the same PTT $\mathcal{A}$ that $\mathcal{S}$ uses internally; it takes as input a uniformly random randomness tape and a tape for simulated honest party randomness and inputs sampled in such a way that these *randomness and inputs pairs are distributed as in $\mathcal{S}$. $\mathcal{Z}$ only has access to the same input/output tapes of $\mathcal{S}^{\mathtt{U}}$ that it can access for $\mathcal{S}$.*

We now also define what we mean by execution consistency. Intuitively, we want the randomness and inputs for simulated honest parties output by an über simulator $\mathcal{S}^{\mathtt{U}}$ to be consistent with the transcripts of real protocol execution.

Definition 6 (Execution Consistency). *We say that the PPT iTM $\mathcal{S}^{\mathtt{U}}$ is* Execution-consistent *if no PPT iTM $\mathcal{Z}$ can distinguish the views of*

1. *$\mathcal{F} \circ \mathcal{S}^{\mathtt{U}}, (x_{h_1}, R_{h_1}), \ldots, (x_{h_k}, R_{h_k}) \xleftarrow{\$} \mathcal{S}^{\mathtt{U}}$: An ideal execution $\mathcal{F}$ with $\mathcal{S}^{\mathtt{U}}$ where $\mathcal{S}^{\mathtt{U}}$ interacts with a copy of $\mathcal{A}$ and with global setup ideal functionalities functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_r$, outputting $(x_{h_1}, R_{h_1}), \ldots, (x_{h_k}, R_{h_k})$ on its special "control output tape" (which is unavailable to $\mathcal{Z}$);*
2. *Executing Π with adversary $\mathcal{A}$ and honest parties $\mathcal{P}_1, \ldots, \mathcal{P}_k$ running on randomness and input pairs $(x_{h_1}, R_{h_1}), \ldots, (x_{h_k}, R_{h_k})$ with global setup ideal functionalities functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_r$,*

where $\mathcal{A}$ is a PPT iTM.

For any PPT iTM $\mathcal{S}^{\mathtt{U}}$ with the input and output tapes defined above, we say that $\mathcal{S}^{\mathtt{U}}$ is an über simulator if it is simulation- and execution-consistent.

Definition 7 (Über Simulator). *We say that the PPT iTM $\mathcal{S}^{\mathtt{U}}$ is an über simulator if there exist input tapes for randomness, simulated honest party inputs such that $\mathcal{S}^{\mathtt{U}}$ is both simulation- and execution-consistent according to Definitions 5 and 6 for any PPT environment $\mathcal{Z}$ and adversary $\mathcal{A}$.*

In Appendix D we provide an example that gives rise to an über simulator.

3.3 Input-aware simulation for existing protocols.

As we outlined in the introduction it is not necessary for each UC-secure protocol to additionally define an über simulator. We therefore now define a restricted class of protocols for which $\mathcal{S}^{\mathtt{U}}$ can be obtained trivially. In order to do that, we assume that, for a protocol Π that UC-realizes $\mathcal{F}$ in the $\mathcal{F}_1, \ldots, \mathcal{F}_r$-hybrid model where all $\mathcal{F}_1, \ldots, \mathcal{F}_r$ are global functionalities with a simulator $\mathcal{S}$, there exists a randomness tape generation function `GenRand` (that generates the randomness input tape for $\mathcal{S}$) as follows:

Function $\mathtt{GenRand}(1^\kappa, R_{h_1}, \ldots, R_{h_k}, x_{h_1}, ..., x_{h_k})$**:** this PPT function has as inputs the security parameter κ, honest party randomness $R_{h_1}, \ldots, R_{h_k}$, honest party inputs $x_{h_1}, \ldots, x_{h_k}$ and outputs a randomness input tape T for $\mathcal{S}$ such that the following properties hold for any PPT iTM $\mathcal{Z}$:

1. $\mathcal{F} \circ \mathcal{S}$ (An ideal execution of $\mathcal{F}$ with $\mathcal{S}$ taking as input an uniformly random randomness tape) is indistinguishable from $\mathcal{F} \circ \mathcal{S}(T)$ (An ideal execution of $\mathcal{F}$ with $\mathcal{S}$ taking as input tape T); and

2. An execution of Π with $\mathcal{A}$ and honest parties $\mathcal{P}_{h_1}, \dots, \mathcal{P}_{h_k}$ taking input/randomness $(x_{h_1}, R_{h_1}), \dots, (x_{h_k}, R_{h_k})$ is indistinguishable from $\mathcal{F} \circ \mathcal{S}(T)$ (An ideal execution of $\mathcal{F}$ with $\mathcal{S}$ taking as input tape T).

It turns out it is possible to easily adapt most existing simulators $\mathcal{S}$ in order to obtain a function `GenRand` with the above property. Most simulators basically run simulated honest parties that execute the protocol with random inputs and randomness, making it easy to parameterize these simulators through their randomness tapes in order to make them use specific randomness and inputs (fed externally) for simulated honest parties. In the case of simulators that run with hard-coded inputs for simulated honest parties, a similar idea can be achieved by modifying them to obtain these inputs from their randomness tapes. Moreover, notice that since an execution of $\mathcal{S}$ without honest party inputs is already known to be indistinguishable from a real world simulation, it follows in most cases that parameterizing $\mathcal{S}$ with simulated honest party inputs that are possibly identical to those in the real world is indistinguishable from the usual execution with $\mathcal{S}$.
Obtaining $\mathcal{S}^{\mathtt{U}}$ from a simulator $\mathcal{S}$ with `GenRand`: We now construct $\mathcal{S}^{\mathtt{U}}$ for a protocol Π that UC-Realizes $\mathcal{F}$ with an original simulator $\mathcal{S}$ as follows: Given the simulator $\mathcal{S}$ and corresponding function `GenRand`, $\mathcal{S}^{\mathtt{U}}$ takes the inputs $x_{h_1}, \dots, x_{h_k}$ and randomness $R_{h_1}, \dots, R_{h_k}$[7] from the honest parties on its input tapes and runs $\mathtt{GenRand}(1^\kappa, R_{h_1}, \dots, R_{h_k}, x_{h_1}, ..., x_{h_k})$ to obtain T. Then $\mathcal{S}^{\mathtt{U}}$ runs a copy of $\mathcal{S}$ with randomness input T. $\mathcal{S}^{\mathtt{U}}$ then forwards all queries between $\mathcal{F}$, $\mathcal{Z}$, a copy of the adversary $\mathcal{A}$, global setup ideal functionalities $\mathcal{F}_1, \dots, \mathcal{F}_r$ and $\mathcal{S}$. In the end, $\mathcal{S}^{\mathtt{U}}$ outputs $(x_{h_1}, R_{h_1}), \dots, (x_{h_k}, R_{h_k})$ on the special output tape. In order to do this, we also assume that instead of running an internal copy of $\mathcal{A}$ it receives all queries from $\mathcal{A}$ (including messages to simulated honest parties and setup ideal functionalities) externally, as well as sending answers to those queries out through the same interface.

Proposition 1. *Given a PPT simulator $\mathcal{S}$ for a protocol Π that UC-realizes $\mathcal{F}$ in the $\mathcal{F}_1, \dots, \mathcal{F}_r$-hybrid model where all $\mathcal{F}_1, \dots, \mathcal{F}_r$ are global functionalities for which a poly-time computable function* `GenRand` *as defined above exists, then the aforementioned $\mathcal{S}^{\mathtt{U}}$ is an über simulator for Π.*

Proof. In order for this construction of $\mathcal{S}^{\mathtt{U}}$ to be a über simulator according to Definition 7, it must both simulation and execution-consistent according to Definitions 5 and 6.

First, we will show that $\mathcal{S}^{\mathtt{U}}$ is simulation-consistent according Definitions 5, which amounts to showing that its internal copy of $\mathcal{S}$ has the same view of $\mathcal{S}$ operating with an uniformly random randomness input tape, an environment $\mathcal{Z}$, an ideal functionality $\mathcal{F}$ and its own copy of $\mathcal{A}$. Notice that all communication to/from $\mathcal{Z}$ (as well as $\mathcal{F}_1, \dots, \mathcal{F}_r$ and $\mathcal{A}$) and $\mathcal{S}$ is simply forwarded by $\mathcal{S}^{\mathtt{U}}$ to/from $\mathcal{S}$. Notice that, since all $\mathcal{F}_1, \dots, \mathcal{F}_r$ are global functionalities,

[7] If honest randomness $R_{h_1}, \dots, R_{h_k}$ is not given, sample all values uniformly at random

$\mathcal{S}$ does not internally simulate local version of these ideal functionalities, instead forwarding requests to them and deciding what to forward back to $\mathcal{A}$. By the properties of `GenRand`, simulating honest parties with tape T produced by $\texttt{GenRand}(1^\kappa, R_{h_1}, \ldots, R_{h_k}, x_{h_1}, ..., x_{h_k})$ is equivalent to using a uniformly random randomness input tape. Hence, $\mathcal{S}^{\mathtt{U}}$ is simulation-consistent, since its internal copy of $\mathcal{S}$ has the same view as in its normal operation, being able to simulate an ideal execution with $\mathcal{F}$ that is indistinguishable from the real world execution with Π and $\mathcal{A}$ (because $\mathcal{S}$ has this property).

In order to see why $\mathcal{S}^{\mathtt{U}}$ is also execution-consistent, notice that `GenRand` by definition guarantees that an execution of $\mathcal{S}$ with randomness tape T obtained form executing $\texttt{GenRand}(1^\kappa, R_{h_1}, \ldots, R_{h_k}, x_{h_1}, ..., x_{h_k})$ is indistinguishable from an execution of Π with $\mathcal{A}$ and honest parties $\mathcal{P}_{h_1}, \ldots, \mathcal{P}_{h_k}$ taking input/randomness $(x_{h_1}, R_{h_1}), \ldots, (x_{h_k}, R_{h_k})$. Hence, since we already know that the $\mathcal{S}^{\mathtt{U}}$'s internal copy of $\mathcal{S}$ has an identical view as in its original operation as shown above, it follows that an execution of $\mathcal{S}^{\mathtt{U}}$ taking as input randomness and input pairs $(x_{h_1}, R_{h_1}), \ldots, (x_{h_k}, R_{h_k})$ for simulated honest parties is indistinguishable from an execution of Π with $\mathcal{A}$ and honest parties $\mathcal{P}_{h_1}, \ldots, \mathcal{P}_{h_k}$ taking the same randomness and input pairs $(x_{h_1}, R_{h_1}), \ldots, (x_{h_k}, R_{h_k})$. Hence, $\mathcal{S}^{\mathtt{U}}$ is execution-consistent, which completes the proof. $\square$

Ideal Functionalities with embedded Über Simulation Assume that Π is a protocol which UC-securely implements the functionality $\mathcal{F}$ using a simulator $\mathcal{S}$. As it was already outlined in Section 1.1 we cannot hope that a black-box approach which only focuses on inputs, randomness and messages of a protocol Π allows us to equip $\mathcal{F}$ with the interfaces from Section 2.3, unless Π is adaptively secure. Instead, we will additionally assume that there exists an über simulator $\mathcal{S}^{\mathtt{U}}$ for the protocol Π and consider that interfaces $\textbf{Input}_i$, $\textbf{Output}_i^{(\tau)}$ and **nmf** of functionality wrapper $\mathcal{F}^{\mathtt{V}}$ will allow us access to a internal copy of $\mathcal{S}^{\mathtt{U}}$ run by $\mathcal{F}^{\mathtt{V}}$. $\mathcal{F}^{\mathtt{V}}$ allows $\mathcal{S}^{\mathtt{U}}$ to directly query global setup functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_n$.

4 Authentication Functionalities

In this section, we define authentication functionalities that will serve as building blocks for our compiler. Our functionalities allow for a set of parties to jointly authenticate messages but do *not* deliver these messages themselves. Later on, a verifier can check that a given message has indeed been authenticated by a given set of parties, meaning that they have received this message through a channel and agree on it. More interestingly, we introduce a functionality that allows for a set of parties to jointly authenticate *private* messages that they do not know (except in encrypted form) as well as inputs and randomness (which they also only know in encrypted form). Later on, if a message is revealed (*e.g.* by the sender) or an input is opened, a verifier can check that it corresponds to a given secret value previously authenticated by a given set of parties.

As opposed to classical point-to-point or broadcast authenticated channels, our functionalities do not deliver messages to the set of receiving parties and consequently do not ensure consensus. These functionalities come into play in our framework as they allow for verifiers to check that all parties who executed a protocol agree on certain parts of the transcript (that might contain private messages) regardless of how the messages in the transcript have been obtained. Having the parties agree on which messages have been sent limits the adversary's power to generate an alternative transcript aiming at forging a proof that the protocol reached a different outcome, which itself is highly related to Definition 4 from the previous section. Decoupling message authentication from delivery allows for a cleaner model of non-interactive verification, where a verifier may obtain a proof containing an authenticated protocol transcript at any point after protocol execution itself (*i.e.* after messages are exchanged).

Functionality $\mathcal{F}_{\mathsf{PJAuth}}$ (with tokens)

$\mathcal{F}_{\mathsf{PJAuth}}$ interacts with a set of authenticating parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$, a set of public verifiers $\mathcal{V}$ and an ideal adversary $\mathcal{S}$, who is allowed to corrupt a set $I \subset [n]$ of parties where $|I| \leq k$ for a fixed $k < n$. $\mathcal{F}_{\mathsf{PJAuth}}$ maintains an initially empty list $\mathcal{L}$, proceeding as follows:

Message Input: Upon receiving a message $(\textsc{Input}, sid, ssid, \mathcal{P}_i, m)$ from a party $\mathcal{P}_i \in \mathcal{P}$, send $(\textsc{Input}, sid, ssid, \mathcal{P}_i, m)$ to $\mathcal{S}$. Upon receiving $(\textsc{Auth-Token}, sid, ssid, \mathcal{P}_i, m, \sigma_i)$ from $\mathcal{S}$, check that no such message was received previously, otherwise output an error message and halt. Send $(\textsc{Auth-Token}, sid, ssid, \mathcal{P}_i, m, \sigma_i)$ to $\mathcal{P}_i$ and ignore further $\textsc{Input}$ messages with the same *ssid*.

Joint Authentication: Upon receiving a message $(\textsc{Auth}, sid, ssid, \mathcal{P}_i, m)$ from a party $\mathcal{P}_j \in \mathcal{P}$, send $(\textsc{Auth}, sid, ssid, \mathcal{P}_i, m)$ to $\mathcal{S}$. Upon receiving $(\textsc{Auth-Token}, sid, ssid, \mathcal{P}_i, m, \sigma_j)$ from $\mathcal{S}$, check that no such message was received previously, otherwise output an error message and halt. Send $(\textsc{Auth-Token}, sid, ssid, \mathcal{P}_i, m, \sigma_j)$ to $\mathcal{P}_j$. If a message $(\textsc{Input}, sid, ssid, \mathcal{P}_i, m)$ has been received from $\mathcal{P}_i \in \mathcal{P}$ and $(\textsc{Auth}, sid, ssid, \mathcal{P}_i, m)$ has been received from all parties $\mathcal{P}_j \in \mathcal{P}$ for $j \neq i$, add $(sid, ssid, \mathcal{P}_i, m, \sigma_1, \dots, \sigma_n)$ to $\mathcal{L}$.

Public Verification: Upon receiving $(\textsc{Verify}, sid, ssid, \mathcal{P}_i, m, \sigma_1, \dots, \sigma_n)$ from a party $\mathcal{V}_i \in \mathcal{V}$, if $(sid, ssid, \mathcal{P}_i, m, \sigma_1, \dots, \sigma_n) \in \mathcal{L}$, set $v = 1$, else set $v = 0$. Send $(\textsc{Verify}, sid, ssid, \mathcal{P}_i, m, v)$ to $\mathcal{V}_i$.

Fig. 4. Public Joint Authentication Functionality $\mathcal{F}_{\mathsf{PJAuth}}$ (with tokens).

4.1 Public Joint Authentication

First, we focus on the simpler case of authenticating public messages, which can be known by all parties participating in the joint authentication procedure. In this case, the *sender* starts by providing a message and *ssid* pair to the functionality and joint authentication is achieved after each of the other parties sends the same pair back to the functionality. This can be achieved by a simple protocol where all parties sign each message received from each other party in each round, sending the resulting signatures to all other parties. A message is considered authenticated if it is signed by all parties. Notice that this protocol does not ensure consensus and can easily fail if a single party does not provide a valid signature on a single message, which an adversary corrupting any party (or the network) can always cause. However, this failure is captured in the functionality and follows the idea of decoupling message delivery from authentication. Functionality $\mathcal{F}_{\mathsf{PJAuth}}$ is described in Figure 4.

4.2 Secret Joint Authentication

Departing from functionality $\mathcal{F}_{\mathsf{PJAuth}}$ capturing the case of public communication, we will define a functionality $\mathcal{F}_{\mathsf{PSAuth}}$ (described in Figure 5), which will capture the case of communication through private channels. This functionality works similarly to $\mathcal{F}_{\mathsf{PJAuth}}$, allowing parties to jointly authenticate messages received through private channels to which they have access. However, it also allows for *bureaucrat* parties who observe the encrypted communication (but do not see plaintext messages) over the private channel to jointly authenticate a *committed* version of such plaintext messages. If a private message is revealed by its sender (or one of its receivers) at a later point, $\mathcal{F}_{\mathsf{PSAuth}}$ allows for third parties (including the bureaucrats that did not see the plaintext message before) to verify that this message is indeed the one that was jointly authenticated. As in the case of $\mathcal{F}_{\mathsf{PJAuth}}$, $\mathcal{F}_{\mathsf{PSAuth}}$ does not aid in communicating messages or authentication information in any way, reflecting its nature as a pure joint authentication functionality where all communication duties are left to the parties (or another protocol using $\mathcal{F}_{\mathsf{PSAuth}}$).

In order to capture the different actions of each party it interacts with, $\mathcal{F}_{\mathsf{PSAuth}}$ is parameterized by the following (sets of) parties: a party $\mathcal{P}_{sender}$ that is allowed to input messages to be jointly authenticated; a set of parties $\mathcal{P}$ who can read input messages given by $\mathcal{P}_{sender}$ and jointly authenticate them; a set of bureaucrats $\mathcal{B}$ who do not see the message but jointly authenticate that $\mathcal{P}_{sender}$ has sent a certain (still unknown) committed message to the parties $\mathcal{P}$. Notice that $\mathcal{F}_{\mathsf{PSAuth}}$ does not aid in delivering the message input by $\mathcal{P}_{sender}$ either to parties $\mathcal{P}_i \in \mathcal{P}$ in plaintext message form nor to bureaucrats in committed form. Moreover, $\mathcal{F}_{\mathsf{PSAuth}}$ does not aid in sending notifications about sent messages nor joint authentication information to any party. The responsibility for sending messages (in plaintext or committed form) lies with $\mathcal{P}_{sender}$, while the responsibility for notifying any other party that plaintext verification is possible lies with $\mathcal{P}_{sender}$ or parties $\mathcal{P}_i \in \mathcal{P}$, who are the only parties who can retrieve the message that was jointly authenticated. The basic idea for realizing $\mathcal{F}_{\mathsf{PSAuth}}$ is using a signature scheme (captured by $\mathcal{F}_{\mathsf{Sig}}$) and a certified encryption scheme with plaintext verification (captured by $\mathcal{F}_{\mathsf{CPKEPV}}$), *i.e.* an encryption scheme with two crucial properties: (1) An encrypting party is guaranteed to encrypt a message that can only be opened by the intended receiver (*i.e.* it is possible to make sure the public-key used belongs to the intended receiver of the encrypted messages); (2) Both encrypting and decrypting parties can generate publicly verifiable proofs

Functionality $\mathcal{F}_{\mathsf{PSAuth}}$ (with tokens)

$\mathcal{F}_{\mathsf{PSAuth}}$ interacts with a special party $\mathcal{P}_{sender}$, a set of authenticating parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, a set of bureaucrats $\mathcal{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_b\}$, a set of public verifiers $\mathcal{V}$ (s.t. $\mathcal{B} \subset \mathcal{V}$) and an ideal adversary $\mathcal{S}$, who is allowed to corrupt a set $I \subset \{\mathcal{P} \cup \mathcal{B}\}$ where $|I| \leq k$ for a fixed $k < n + b$. $\mathcal{F}_{\mathsf{PSAuth}}$ maintains an initially empty list $\mathcal{L}$, proceeding as follows:

Message Input: Upon receiving a message $(\textsc{Input}, sid, ssid, \mathcal{P}_{sender}, m)$ from $\mathcal{P}_{sender}$ ignore further $\textsc{Input}$ messages with the same $ssid$. Additionally, send $(\textsc{Auth-Token}, sid, ssid, \mathcal{P}_{sender}, m, \sigma_{sender})$ to $\mathcal{P}_{sender}$ and forward $(\textsc{Input}, sid, ssid, \mathcal{P}_{sender})$ to $\mathcal{S}$.

Joint Authentication: Upon receiving a message $(\textsc{Auth}, sid, ssid, \mathcal{P}_{sender}, m)$ from a party $\mathcal{P}_i \in \mathcal{P}$ (resp. $(\textsc{Blind-Auth}, sid, ssid, \mathcal{P}_{sender})$ from a bureaucrat $\mathcal{B}_j \in \mathcal{B}$), if a message $(\textsc{Input}, sid, ssid, \mathcal{P}_{sender}, m)$ has been received from $\mathcal{P}_{sender}$, forward the AUTH (resp. $\mathsf{Blind-Auth}$) to $\mathcal{S}$. Upon receiving $(\textsc{Auth-Token}, sid, ssid, \mathcal{P}_{sender}, m, \sigma_i)$ (resp. $(\textsc{Auth-Token}, sid, ssid, \mathcal{P}_{sender}, \hat{\sigma}_j)$) from $\mathcal{S}$, check that no such message was received previously, otherwise output an error message and halt. Send $(\textsc{Auth-Token}, sid, ssid, \mathcal{P}_{sender}, m, \sigma_i)$ to $\mathcal{P}_i$ (resp. $(\textsc{Auth-Token}, sid, ssid, \mathcal{P}_{sender}, \hat{\sigma}_j)$ to $\mathcal{B}_j$). Additionally, forward $(\textsc{Auth}, sid, ssid, \mathcal{P}_{sender})$ (resp. $(\textsc{Blind-Auth}, sid, ssid, \mathcal{P}_{sender})$) to $\mathcal{S}$. If messages $(\textsc{Auth}, sid, ssid, \mathcal{P}_{sender}, m)$ has been received from all parties $\mathcal{P}_i \in \mathcal{P}$ and messages $(\textsc{Blind-Auth}, sid, ssid, \mathcal{P}_{sender})$ has been received from all bureaucrats $\mathcal{B}_j \in \mathcal{B}$, add $(sid, ssid, \mathcal{P}_{sender}, m, \sigma_{sender}, \sigma_1, \ldots, \sigma_n, \hat{\sigma}_1, \ldots, \hat{\sigma}_b, \perp)$ to $\mathcal{L}$.

Public Verification: Upon receiving $(\textsc{Verify}, sid, ssid, \mathcal{P}_{sender}, m, \sigma_{sender}, \sigma_1, \ldots, \sigma_n, \hat{\sigma}_1, \ldots, \hat{\sigma}_b)$ from a party $\mathcal{V}_i \in \mathcal{V}$, if $(sid, ssid, \mathcal{P}_{sender}, m, \sigma_{sender}, \sigma_1, \ldots, \sigma_n, \hat{\sigma}_1, \ldots, \hat{\sigma}_b,) \in \mathcal{L}$, set $v = 1$, else set $v = 0$. Send $(\textsc{Verify}, sid, ssid, \mathcal{P}_{sender} m, v)$ to $\mathcal{V}_i$.

Fig. 5. Secret Joint Authentication Functionality $\mathcal{F}_{\mathsf{PSAuth}}$ (with tokens).

that a certain message was contained in a given ciphertext. The private channel itself is realized by encrypting messages under the encryption scheme, while joint authentication is achieved by having all parties in $\mathcal{P}$ (including the sender) and bureaucrats in $\mathcal{B}$ sign the resulting ciphertext. In order to obtain efficiency, a joint public/secret key pair is generated for each set of receivers, in such a way that the same ciphertext can be decrypted by all the receivers holding the corresponding joint secret key. Later on, if any party in $\mathcal{P}$ (including the sender) wishes to start the verification procedure to prove that a certain message was indeed contained in the ciphertext associated with a given $ssid$, it recovers the plaintext message and a proof of plaintext validity from the ciphertext and sends those to one or more verifiers. With these values, any party can first verify that the ciphertext that was sent indeed corresponds to that message due to the plaintext verification property of the encryption scheme and then verify that it has been jointly authenticated by checking that there exist valid signatures on that ciphertext by all parties in $\mathcal{P}$ and bureaucrats in $\mathcal{B}$. The details of the construction are described in Appendix C.

Functionality $\mathcal{F}_{\mathsf{IRAuth}}$

$\mathcal{F}_{\mathsf{IRAuth}}$ interacts with a set of n parties $\mathcal{P}$, a set of public verifiers $\mathcal{V}$ and an ideal adversary $\mathcal{S}$ who is allowed to corrupt a set $I \subset \mathcal{P}$ where $|I| \leq k$ for a fixed $k < n$. $\mathcal{F}_{\mathsf{IRAuth}}$ maintains an initially empty list $\mathcal{L}$, proceeding as follows:

Message Input: Upon receiving a message $(\textsc{Input}, sid, \mathcal{P}_i, m)$ from a party $\mathcal{P}_i \in \mathcal{P}$ ignore further $\textsc{Input}$ messages from $\mathcal{P}_i$.

Joint Authentication: Upon receiving a message $(\textsc{Blind-Auth}, sid, \mathcal{P}_i, \mathcal{P}_j)$ from a party $\mathcal{P}_j \in \mathcal{P}$, $j \neq i$, if a message $(\textsc{Input}, sid, \mathcal{P}_i, m)$ has been received from $\mathcal{P}_i$ and a message $(\textsc{Blind-Auth}, sid, \mathcal{P}_i, \mathcal{P}_j)$ has been received from all parties $\mathcal{P}_j \in \mathcal{P} \backslash \{\mathcal{P}_i\}$, add $(sid, \mathcal{P}_i, m, \perp)$ to $\mathcal{L}$.

Start Verification: Upon receiving a message $(\textsc{Start-Verify}, sid, \mathcal{P}_i, m)$ from $\mathcal{P}_i$, if there exists an entry $(sid, \mathcal{P}_i, m, \perp)$ in $\mathcal{L}$, update it to $(sid, \mathcal{P}_i, m, \textsc{verify})$.

Public Verification: Upon receiving $(\textsc{Verify}, sid, \mathcal{P}_i, m)$ from a party $\mathcal{V}_i \in \mathcal{V}$, if ($sid, \mathcal{P}_i, m, \textsc{verify}) \in \mathcal{L}$, set $v = 1$, else set $v = 0$. Send $(\textsc{Verify}, sid, \mathcal{P}_i, m)$ to $\mathcal{S}$ and, if $\mathcal{S}$ answers with $(\textsc{Proceed}, sid, ssid, m)$, send $(\textsc{Verify}, sid, m, v)$ to $\mathcal{V}_i$. Otherwise, send $(\textsc{Verify}, sid, m, 0)$ to $\mathcal{V}_i$.

Fig. 6. Input and Randomness Authentication Functionality $\mathcal{F}_{\mathsf{IRAuth}}$.

4.3 Authenticating Inputs and Randomness

To provide an authentication of inputs and randomness we adapt the functionality $\mathcal{F}_{\mathsf{PSAuth}}$, as the desired capabilities are like a message authentication without a receiver. Alternatively, one could express it also in the context of non-interactive multi-receiver commitments. In Figure 6 we present a functionality which implements this. The functionality works in the sense of cir of Definition 4, as it allows each party to commit to a unique string (for input and randomness of the protocol) towards all parties. We refer readers who are interested in an implementation of $\mathcal{F}_{\mathsf{IRAuth}}$ to Section 4.2, as any realization of $\mathcal{F}_{\mathsf{PSAuth}}$ can easily be adapted to $\mathcal{F}_{\mathsf{IRAuth}}$. Notice that $\mathcal{F}_{\mathsf{IRAuth}}$ can be instantiated from n instances of $\mathcal{F}_{\mathsf{PSAuth}}$ such that, for each $\mathcal{P}_i \in \mathcal{P}$ interacting with $\mathcal{F}_{\mathsf{IRAuth}}$, there is an instance $\mathcal{F}^i_{\mathsf{PSAuth}}$ where $\mathcal{P}_i$ acts as $\mathcal{P}_{sender}$, the set of bureaucrats $\mathcal{B}^i$ of $\mathcal{F}^i_{\mathsf{PSAuth}}$ is equal to the set $\mathcal{P}$ of $\mathcal{F}_{\mathsf{IRAuth}}$ and the set $\mathcal{P}$ of $\mathcal{F}^i_{\mathsf{PSAuth}}$ only contains $\mathcal{P}_i$.

5 Compilation for Input-Revealing Protocols

We now show how to compile the protocols from Section 2.1 into non-interactively verifiable counterparts. To achieve this we will in some cases only have to rely on a signature functionality, whereas a compiler for the *weakest* protocols according to Definition 4 needs rather strong additional tools such as the authentication functionalities from the previous section. In this work we focus on protocols according to Definition 4 and as such there are 8 different combinations of parameters (ν, μ) for (ν, CIR, μ)-collusion resistant protocols which we might consider. Furthermore, according to Definition 2 we might either have public or private verifier registration, which in total yields 16 different definitions. To avoid redundancy we now outline how to achieve the respective verifiability in each setting and a thorough analysis of a general technique that works for any publicly verifiable (ν, CIR, μ)-collusion resistant protocols. We simplify notation by just assuming the existence of a single verifier $\mathcal{V}$.

5.1 How to make Protocols Verifiable

We now describe how to combine all the introduced building blocks and notation from the previous sections to make a protocol verifiable. More specifically, we take a (ν, CIR, μ)-collusion resistant protocol Π that UC realizes an ideal functionality $\mathcal{F}$ in the (global) $\mathcal{F}_1, \ldots, \mathcal{F}_r$-hybrid model with über simulator $\mathcal{S}^{\mathsf{U}}$ and do the following:

1. We describe how to construct a protocol Π^{V} by modifying Π with access to a signature functionality $\mathcal{F}_{\mathsf{Sig}}$, a key registration functionality $\mathcal{F}_{\mathsf{Reg}}$ and authentication functionalities $\mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{IRAuth}}$. We will furthermore require that we can replace the hybrid functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_r$ used in Π with verifiable counterparts.
2. We then show that Π^{V} UC-realizes $\mathcal{F}^{\mathsf{V}}[\mathcal{F}]$ as described in Section 3.3 in the (global) $\mathcal{F}_1, \ldots, \mathcal{F}_r$-hybrid by constructing an explicit simulator $\mathcal{S}^{\mathsf{V}}$.

For each of the different choices of ν and μ there is a different way how Π must be compiled to Π^{V} and we will not describe all 8 different possibilities (and prove them secure) for the sake of conciseness. We will instead now explain on a high level which transformations are necessary, and will then explain the proof technique for the general case of making a publicly verifiable version of any (ν, CIR, μ)-collusion resistant protocol.

Protocol Compilation - The Big Picture. In order to verify we let the verifier $\mathcal{V}$ simulate each such party whose output shall be checked and which participated in an instance of Π. This check is done locally, based on the inputs, randomness and messages related to such a party (and/or other parties) which $\mathcal{V}$ obtains for this process. In case of public verifier registration we assume that a bulletin board is available which holds the protocol transcript, whereas in case of private registration the verifier contacts one of the protocol parties to obtain a transcript which it can then verify non-interactively. We want to stress that the Bulletin Board which may contain the protocol transcript *do not have to be used to exchange messages during the actual protocol run.*

In Π we assume that messages can either be exchanged secretly between two parties or via a broadcast channel. Furthermore, parties may send messages to hybrid functionalities or receive them from these. An adversary may now be able to replace certain parts of the protocol transcript. As long as we assume that a protocol is (ν, CIR, μ)-collusion resistant and constrain his ability to maul the protocol transcript to those parts permitted by the definition, the overall construction achieves verifiability. We now explain, on a high level, the modifications to Π for the different values of μ, ν:

$\mu = \mathsf{ncmes}$: Here the adversary is allowed to replace all messages by any party at his will, and messages are just exchanged as in Π.

$\mu = \mathsf{chsmes}$: Before the protocol begins, each $\mathcal{P}_i$ first generates a signing key with $\mathcal{F}_{\mathsf{Reg}}$ registers its signing key with $\mathcal{F}_{\mathsf{Reg}}$. Whenever a party $\mathcal{P}_i$ sends a message $\mathtt{m}_{i,j}^{(\tau,\rho)}$ to $\mathcal{P}_j$ it then uses $\mathcal{F}_{\mathsf{Sig}}$ to authenticate $\mathtt{m}_{i,j}^{(\tau,\rho)}$ with a signature $\sigma_{i,j}^{(\tau,\rho)}$. In such a case, $\mathcal{V}$ will later be able to correctly verify exactly those messages of the transcript that were sent by honest parties, as $\mathcal{A}$ might fake messages and signatures sent by dishonest parties after the fact.

$\mu = \mathsf{chmes}$: In this setting, each message that is either sent or received by an honest party must remain unaltered. Each party will do the same as in the case where $\mu = \mathsf{chsmes}$, but we additionally require that whenever a party $\mathcal{P}_i$ receives a message $\mathtt{m}_{j,i}^{(\tau,\rho)}$ from $\mathcal{P}_j$ then it then uses $\mathcal{F}_{\mathsf{Sig}}$ to authenticate $\mathtt{m}_{j,i}^{(\tau,\rho)}$ with a signature $\sigma(R)_{j,i}^{(\tau,\rho)}$. Now $\mathcal{V}$ can establish for each message of the protocol if both sender and receiver signed the same message, which will allow $\mathcal{A}$ to only alter those messages that were both sent and received by dishonest parties.

$\mu = \mathsf{cmes}$: We now also require that the dishonest parties cannot replace their messages before verification. To achieve this, we use $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}$ as defined in Section 4 which the parties must now use in order to register their private message exchange. These functionalities $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}$ can then be used by $\mathcal{V}$ in order to validate an obtained transcript.

$\nu = \mathsf{ncir}$: Based on each $\mathcal{P}_i$ setting up a key with $\mathcal{F}_{\mathsf{Sig}}$ and registering it with $\mathcal{F}_{\mathsf{Reg}}$ let each party sign both its input x_i and its randomness r_i using $\mathcal{F}_{\mathsf{Sig}}$ before sending it in **Activate Verification**, which means that $\mathcal{V}$ only accepts such signed values which it can verify via $\mathcal{F}_{\mathsf{Sig}}$. $\mathcal{A}$ can later replace the pairs (x_j, r_j) of dishonest parties $\mathcal{P}_j$ by generating different signatures.

$\nu = \mathsf{cir}$: The parties will use the available functionality $\mathcal{F}_{\mathsf{IRAuth}}$ to authenticate their inputs and randomness initially. Later, $\mathcal{V}$ can use $\mathcal{F}_{\mathsf{IRAuth}}$ to check validity of the revealed x_i, r_i which it obtained for verification.

Hybrid Functionalities: As mentioned above we replace the auxiliary functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_r$ with NIV counterparts, i.e. with functionalities $\mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_r^{\mathtt{V}}$ that have the same interfaces as defined in Definition 3. If we intend to achieve public verifiability then each such $\mathcal{F}_q^{\mathtt{V}}$ must also be publicly verifiable, whereas in the case of private verifiability either type of functionality is fine. For any such $\mathcal{F}_q^{\mathtt{V}}$ we can then establish if a certain message $\mathtt{mres}_{q,i}$ was indeed sent to $\mathcal{P}_i$ or not. If $\mathcal{F}_q^{\mathtt{V}}$ does also reveal inputs, then we can furthermore test if $\mathtt{mres}_{i,q}$ as claimed to be sent by $\mathcal{P}_i$ was indeed received by the respective party.

We now show how to formally embed the aforementioned transformations into a protocol in order to achieve non-interactive UC verifiability.

Protocol $\mathit{\Pi}^{\mathtt{V}}$

The protocol $\mathit{\Pi}^{\mathtt{V}}$ is parameterized by a protocol $\mathit{\Pi}$ with next message function `nmes` and output function `out` as defined in Section 2.1. $\mathit{\Pi}^{\mathtt{V}}$ uses functionalities $\mathcal{F}_{\mathsf{BB}}, \mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{IRAuth}}$ as well as global hybrid functionalities $\mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_r^{\mathtt{V}}$ where $\mathit{\Pi}$ has used possibly non-verifiable versions thereof. All of these functionalities are available to the verifiers $\mathcal{V}$. We set up one copy $\mathcal{F}_{\mathsf{PSAuth}}^{(i,j)}$ for any private communication where $\mathcal{P}_i$ is $\mathcal{P}_{sender}$, $\mathcal{P}_j$ acts as authenticating party and all other parties $\mathcal{P} \setminus \{\mathcal{P}_i, \mathcal{P}_j\}$ act as bureaucrats.
Initially, the parties will run any necessary **Initialization** of the functionalities involved such as to e.g. register keys. They then do the following:

Input$_i$: On input x_i party $\mathcal{P}_i$ runs $\mathit{\Pi}$.**Input**$_i$. Afterwards, $\mathcal{P}_i$ sends (Input, $sid, \mathcal{P}_i, (x_i, r_i)$) to $\mathcal{F}_{\mathsf{IRAuth}}$, while each $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$ sends (Blind-Auth, $sid, \mathcal{P}_i, \mathcal{P}_j$).
Compute$^{(\tau)}$: Each $\mathcal{P}_i$ does the following:
1. For every $\rho \in [H_\tau]$, first run the 4 steps of $\mathit{\Pi}$.**Compute**$^{(\tau)}$.
2. If a party $\mathcal{P}_i$ sent a broadcast message m in round ρ, then $\mathcal{P}_i$ sends (Input, $sid, ssid, \mathcal{P}_i, m$) to $\mathcal{F}_{\mathsf{PJAuth}}$ while each $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$ sends (Auth, $sid, ssid, \mathcal{P}_i, m$).
3. If $\mathcal{P}_i$ sent private messages, then for the receiver $\mathcal{P}_j \in \mathcal{P} \setminus \{p_i\}$ of the message $\mathtt{m}_{i,j}^{(\tau,\rho)}$ $\mathcal{P}_i$ sends (Input, $sid, ssid, \mathcal{P}_i, m$) to $\mathcal{F}_{\mathsf{PSAuth}}^{(i,j)}$ while $\mathcal{P}_j$ sends (Auth, $sid, ssid, \mathcal{P}_i, m$) and each remaining party from $\mathcal{P} \setminus \{\mathcal{P}_i, \mathcal{P}_j\}$ sends (Blind-Auth, $sid, ssid, \mathcal{P}_i$).

Output$_i^{(\tau)}$: $\mathcal{P}_i$ does the same as in $\mathit{\Pi}$.**Output**$_i^{(\tau)}$.
Register Verifier: $\mathcal{V}$ sends (Register, sid) to each $\mathcal{F}_q^{\mathtt{V}}$ for $q \in [r]$.
Activate Verification: On input (Activate-Verification, sid, $\mathsf{open\text{-}i}$, $\mathsf{open\text{-}input\text{-}i}$), $\mathcal{P}_i$ does the following:
1. Send (Activate-Verification, sid, 1) to each $\mathcal{F}_q^{\mathtt{V}}$ for $q \in [r]$.
2. If $\mathsf{open\text{-}input\text{-}i} = 1$, then post $x_i, r_i, \mathcal{N}_{\cdot,i}, \hat{\mathcal{M}}_{\cdot,j}$ on $\mathcal{F}_{\mathsf{BB}}$.

Fig. 7. The protocol $\mathit{\Pi}^{\mathtt{V}}$ which makes the (ν, CIR, μ)-collusion resistant protocol $\mathit{\Pi}$ publicly verifiable.

5.2 Public Verification for any (ν, CIR, μ)-collusion resistant protocol

The basic idea of this construction is to turn any (ν, CIR, μ)-collusion resistant protocol into a $(\mathsf{cir}, \mathsf{CIR}, \mathsf{cmes})$-collusion resistant protocol by forcing the adversary to commit to all the corrupted parties' randomness, inputs and messages. While this might be overkill for some protocols (*e.g.* $(\mathsf{ncir}, \mathsf{CIR}, \mathsf{cmes})$-collusion resistant protocols), we focus on the worst case scenario of compiling $(\mathsf{ncir}, \mathsf{CIR}, \mathsf{ncmes})$-collusion resistant protocols, since it is the most challenging. Note that, after making a protocol $(\mathsf{cir}, \mathsf{CIR}, \mathsf{cmes})$-collusion resistant, the protocol execution becomes deterministic and can be verified upon the revealing of the randomness, input and transcript of any party that activates the verification. All the verifier has to do is to execute the protocol's next message function on these randomness and input taking received messages from the transcript.

If a corrupted party who activates verification attempts to cheat by revealing fake values for randomness, input and transcript, it is caught because those values were committed to. Apart from having all parties commit to jointly authenticated versions of their randomness, inputs and transcripts, the protocol we present requires an authenticated bulleting board where this information is posted in the clear if a party activates verification revealing its input and randomness. We remark that *the bulletin board is not necessary* for employing our techniques, since the values revealed for verification can simply be (unreliably) been sent among parties. We use a trusted bulletin board in order to focus on the important aspects of applying our techniques to existing protocols without the distraction of analyzing all corner cases that arise from operating on unreliable verification data. We stress that in these cases no adversary would be able to force verification to succeed for a cheating party or produce a fake proof showing an honest party cheated. Moreover, the overhead of $\mathcal{F}_{\mathsf{PSAuth}}$ and $\mathcal{F}_{\mathsf{PJAuth}}$ can be avoided if instead of a (ν, CIR, μ)-collusion resistant protocol we use as the starting point a $(\nu, \mathsf{CIR}, \mathsf{cmes})$-collusion resistant protocol or at least reduced if we depart from another protocol where some of the messages are naturally fixed (*e.g.* a $(\nu, \mathsf{CIR}, \mathsf{chmes})$-collusion resistant protocol).

Given a (ν, CIR, μ)-collusion resistant protocol $\Pi = (\mathtt{nmes}, \mathtt{out})$ that UC realizes an ideal functionality $\mathcal{F}$ in the (global) $\mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_r^{\mathtt{V}}$-hybrid model with über simulator $\mathcal{S}^{\mathtt{U}}$, we construct a protocol $\Pi^{\mathtt{V}}$ that UC-realizes the publicly verifiable ideal functionality $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$ in the $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{IRAuth}}, \mathcal{F}_{\mathsf{BB}}, \mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_r^{\mathtt{V}}$-hybrid model. Protocol $\Pi^{\mathtt{V}}$ is described in Figures 7 and 8.

Theorem 1. *Let Π be a (ν, CIR, μ)-collusion resistant protocol that UC realizes an ideal functionality $\mathcal{F}$ in the (global) $\mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_r^{\mathtt{V}}$-hybrid model with über simulator $\mathcal{S}^{\mathtt{U}}$. Then $\Pi^{\mathtt{V}}$ UC-realizes the publicly verifiable ideal functionality $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$ in the $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{IRAuth}}, \mathcal{F}_{\mathsf{BB}}, \mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_r^{\mathtt{V}}$-hybrid model.*

Proof. In order to prove Theorem 1 we construct a simulator $\mathcal{S}$ that interacts with environment $\mathcal{Z}$, functionality $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$, global functionalities $\mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_r^{\mathtt{V}}$ and a internal copy of an adversary $\mathcal{A}$ who may corrupt a subset $I \subset \mathcal{P}$ of size at most k while $\mathcal{S}$ will simulate the remaining parties $\overline{I} = \mathcal{P} \setminus I$ as well as the resources used in $\Pi^{\mathtt{V}}$. $\mathcal{S}$ forwards all communication between $\mathcal{A}$ and $\mathcal{Z}$. $\mathcal{S}$ simulates setup functionalities $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{IRAuth}}, \mathcal{F}_{\mathsf{BB}}, \mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_r^{\mathtt{V}}$ exactly as they are described, except for when alternative behavior is described

The rationale in our construction of $\mathcal{S}$ is straightforward: it takes care of simulating the extra interfaces added to $\mathcal{F}$ by $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$ (along with the extra setup functionalities $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{IRAuth}}, \mathcal{F}_{\mathsf{BB}}$), while delegating simulation of the original Π to its über simulator $\mathcal{S}^{\mathtt{U}}$ incorporated into $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. In the Input phase of $\Pi^{\mathtt{V}}$, $\mathcal{S}$ extracts the randomness and input pair of corrupted parties controlled by $\mathcal{A}$ from the simulated $\mathcal{F}_{\mathsf{IRAuth}}$ and forwards them to the $\mathbf{Input}_i$ interface of $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. For the Compute and Output phases of $\Pi^{\mathtt{V}}$, $\mathcal{S}$ forwards all requests from $\mathcal{A}$ to the über simulator $\mathcal{S}^{\mathtt{U}}$ for the original protocol Π through the **nmf** interface of $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. Upon receiving a response from $\mathcal{S}^{\mathtt{U}}$, it forwards it back to $\mathcal{A}$. Apart from forwarding direct communication between $\mathcal{A}$ and simulated honest parties to $\mathcal{S}^{\mathtt{U}}$, it also simulates $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{IRAuth}}, \mathcal{F}_{\mathsf{BB}}$, verifying messages to simulated honest parties that should also be forwarded to $\mathcal{S}^{\mathtt{U}}$ are properly authenticated

Protocol $\Pi^{\mathtt{V}}$ (Continuation)

Verify$_k$: $\mathcal{V}$ on input $k, a, b^{(1)}, \ldots, b^{(G)}$ does the following:

1. For party $\mathcal{P}_j$ check that $x_j, r_j, \mathcal{N}_{\cdot,j}, \mathcal{M}_{\cdot,j}$ are on $\mathcal{F}_{\mathsf{BB}}$. Otherwise output $(\textsc{Cannot-Verify}, sid, j)$.
   For each functionality $\mathcal{F}_q^{\mathtt{V}}$ verify that $\mathcal{N}_{\cdot,i}$ is valid by doing the following:
   - If $\mathcal{F}_q^{\mathtt{V}}$ is Input-Private then send $(\textsc{Verify}, sid, j, b_{j,q}^{(1)}, \ldots, b_{j,q}^{(G)})$ for each $j \in [n]$, where $b_{j,q}^{(1)}, \ldots, b_{j,q}^{(G)}$ are taken from $\mathcal{N}_{\cdot,j}$. If either $\mathcal{F}_q^{\mathtt{V}}$ returns $(\textsc{Verify}, sid, j, B)$ with $B \neq \emptyset$ or $(\textsc{Cannot-Verify}, sid, j)$ then output $(\textsc{Cannot-Verify}, sid, j)$.
   - If $\mathcal{F}_q^{\mathtt{V}}$ is Input-Revealing then instead send $(\textsc{Verify}, sid, j, x_{j,q}, b_{j,q}^{(1)}, \ldots, b_{j,q}^{(G)})$ where $x_{j,q}$ is derived from the protocol execution. If either $\mathcal{F}_q^{\mathtt{V}}$ returns $(\textsc{Verify}, sid, j, f, B)$ with $B \neq \emptyset$, $f = 0$ or if it returns $(\textsc{Cannot-Verify}, sid, j)$ then output $(\textsc{Cannot-Verify}, sid, j)$.
2. Run the protocol Π by simulating $\mathcal{P}_j$ using the next message function `nmes` using $\mathcal{N}_{\cdot,j}, \mathcal{M}_{\cdot,j}$ with input x_j and randomness r_j until an output a can be obtained by the output function `out`. Check for each broadcast message generated for $\mathcal{P}_j$ by `nmes` (resp. contained in $\mathcal{M}_{\cdot,j}$) that this message was sent (resp. received) via $\mathcal{F}_{\mathsf{PJAuth}}$ and similarly verify private messages generated for $\mathcal{P}_j$ by `nmes` (resp. contained in $\mathcal{M}_{\cdot,j}$) from (resp. to) $\mathcal{P}_j$ to (resp. from) $\mathcal{P}_i$ via $\mathcal{F}_{\mathsf{PSAuth}}^{(i,j)}$. In case of any inconsistency, output $(\textsc{Cannot-Verify}, sid, k)$.

   Then define $f = 1$ if $a = x_j$ and $f = 0$ otherwise as well as $B = \{\tau \in [G] \mid y^{(\tau)} \neq \mathtt{out}(k, x_k, r_k, s, \rho, \mathcal{M}_{\cdot,k}, \mathcal{N}_{\cdot,k})\}$ and return $(\textsc{Verify}, sid, k, f, B)$.

Fig. 8. The protocol $\Pi^{\mathtt{V}}$ which makes the $(\mathsf{cir}, \mathsf{cmes})$-collusion resistant protocol Π publicly verifiable (continuation).

and later simulating $\mathcal{S}^{\mathtt{U}}$'s response being authenticated by the right functionality as coming from the right simulated honest party. If verification is initiated by $\mathcal{A}$, $\mathcal{S}$ checks that $\mathcal{A}$ has provided correct authentication data according to $\Pi^{\mathtt{V}}$, in which case it activates verification through the **Activate Verification** interface of $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$ (otherwise it does not). If verification is initiated by an honest party, $\mathcal{S}$ obtains from $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$ the randomness and input (r_i, x_i) used by the honest party $\mathcal{P}_i$ who initiated verification and simulates that honest party initiating verification with $(r_i, x_i, \mathcal{N}_{\cdot,i}, \mathcal{M}_{\cdot,i})$ towards $\mathcal{A}$ by simulating these values being posted to $\mathcal{F}_{\mathsf{BB}}$, where $\mathcal{N}_{\cdot,i}, \mathcal{M}_{\cdot,i}$ are generated according to the simulated execution towards $\mathcal{A}$. Finally, $\mathcal{S}$ simulates verification by acting exactly as in $\Pi^{\mathtt{V}}$ and forwarding queries to the **Verify**$_j$ interface of $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. Also, if $\mathcal{A}$ produced incorrect verification data for some of the corrupted parties, $\mathcal{S}$ instructs $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$ to make verification activation queries for the corresponding parties to fail.

In order to see why the simulation with $\mathcal{S}$ is indistinguishable from a real execution of $\Pi^{\mathtt{V}}$, we will first analyze the simulation of the Input, Compute and Output phases. First, $\mathcal{S}$ extracts $\mathcal{A}$'s randomness and inputs from $\mathcal{F}_{\mathsf{IRAuth}}$ and sends them to the $\mathbf{Input}_i$ interface of $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]]$. In the remainder of these phases, $\mathcal{S}$ follows the exact steps of $\Pi^{\mathtt{V}}$ and delegates the simulation of the underlying protocol Π to its über simulator $\mathcal{S}^{\mathtt{U}}$ incorporated into $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. Notice that $\mathcal{S}^{\mathtt{U}}$ is parameterized with the randomness and input from honest parties by definition of $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. Since $\mathcal{S}$ also forwards all communication between $\mathcal{S}^{\mathtt{U}}$, this part of the simulation is indistinguishable from a real execution by $\mathcal{S}^{\mathtt{U}}$'s properties according to Definition 7.

It remains to be shown that a simulation of the **Activate Verification** and $\mathbf{Verify}_k$ phases with $\mathcal{S}$ is also indistinguishable from a real execution of these phases with $\mathcal{A}$. First, notice again that, since $\mathcal{S}^{\mathtt{U}}$ is an über simulator parameterized as discussed before, according to Definition 7 all the transcript $\mathcal{N}_{\cdot,i}, \mathcal{M}_{\cdot,i}$ forwarded between $\mathcal{S}^{\mathtt{U}}$ and $\mathcal{A}$ is consistent with the inputs and randomness (r_i, x_i) obtained from $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. Next, notice that, since the randomness and inputs of all parties are committed to using $\mathcal{F}_{\mathsf{IRAuth}}$ and all messages between corrupted parties controlled by $\mathcal{A}$ and honest parties simulated by $\mathcal{S}$ (with the help of $\mathcal{S}^{\mathtt{U}}$) are authenticated using $\mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{PSAuth}}$, the execution of Π during the Input, Compute and Output phases is equivalent the execution of a $(\mathsf{cir}, \mathsf{CIR}, \mathsf{cmes})$-collusion resistant protocol in the game of Definition 4 (where the parties are not allowed to alter their randomness, input and transcript after the protocol is executed). Notice also that executing the verification procedure of $\Pi^{\mathtt{V}}$ is equivalent to performing the procedures of the challenger in the game of Definition 4. Hence, when $\mathcal{S}$ executes the verification phase by following the steps of $\Pi^{\mathtt{V}}$, it is guaranteed by Definition 4 to arrive at the correct result about the presence of cheating parties (or lack thereof). Since $\mathcal{S}$ either allows verification to succeed or makes it fail according to the checks it performs following the instructions of $\Pi^{\mathtt{V}}$ and those checks detect cheating correctly with all but negligible probability (by Definition 4), that proves the remaining case and concludes our proof.

References

1. Joël Alwen, Rafail Ostrovsky, Hong-Sheng Zhou, and Vassilis Zikas. Incoercible multi-party computation and universally composable receipt-free voting. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 763–780. Springer, Heidelberg, August 2015.
2. Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE Computer Society Press, May 2014.
3. Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017.
4. Carsten Baum, Bernardo David, and Rafael Dowsley. Insured mpc: Efficient secure computation with financial penalties. To appear at Financial Cryptography and Data Security (FC) 2020. Full version available at https://eprint.iacr.org/2018/942.
5. Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439. Springer, Heidelberg, August 2014.

6. Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, April / May 2018.
7. Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. UC commitments for modular protocol design and applications to revocation and attribute tokens. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 208–239. Springer, Heidelberg, August 2016.
8. Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. Uc-secure non-interactive public-key encryption. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 217–233. IEEE Computer Society, 2017.
9. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
10. Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.
11. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.
12. Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key-exchange protocols. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 380–403. Springer, Heidelberg, March 2006.
13. Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 597–608. ACM Press, November 2014.
14. Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. Relaxing chosen-ciphertext security. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 565–582. Springer, Heidelberg, August 2003.
15. Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, Heidelberg, August 2003.
16. Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 13–25. Springer, Heidelberg, August 1998.
17. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.
18. Bernardo David, Rafael Dowsley, and Mario Larangeira. Kaleidoscope: An efficient poker protocol with payment distribution and penalty enforcement. In Sarah Meiklejohn and Kazue Sako, editors, *FC 2018*, volume 10957 of *LNCS*, pages 500–519. Springer, Heidelberg, February / March 2018.
19. Bernardo David, Rafael Dowsley, and Mario Larangeira. ROYALE: A framework for universally composable card games with financial rewards and penalties enforcement. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 282–300. Springer, Heidelberg, February 2019.

20. Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 369–386. Springer, Heidelberg, August 2014.
21. Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 705–734. Springer, Heidelberg, May 2016.
22. Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 418–429. ACM Press, October 2016.
23. Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 25(4):680–722, October 2012.
24. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.
25. David Pointcheval. Chosen-ciphertext security for any one-way cryptosystem. In Hideki Imai and Yuliang Zheng, editors, *PKC 2000*, volume 1751 of *LNCS*, pages 129–146. Springer, Heidelberg, January 2000.

Supplementary Material

A Auxiliary Functionalities

Digital Signatures Ideal Functionality $\mathcal{F}_{\mathsf{Sig}}$. The standard digital signature functionality $\mathcal{F}_{\mathsf{Sig}}$ from [10] is presented in Figure 9. It is also shown in [10] that any EUF-CMA signature scheme UC realizes this functionality.

Key Registration Ideal Functionality $\mathcal{F}_{\mathsf{Reg}}$. The key registration functionality $\mathcal{F}_{\mathsf{Reg}}$ from [12] is presented in Figure 10. This ideal functionality allows parties to register public-keys in such a way that other parties can retrieve such keys with the guarantee that they belong to the party who originally registered them. This functionality will be used as setup for the constructions of certified public-key encryption with plaintext verification and secret joint authentication of Section 4.2.

Bulletin Board Ideal Functionality $\mathcal{F}_{\mathsf{BB}}$. In Figure 11 we describe an authenticated bulletin board functionality which is used throughout this work. Authenticated Bulletin Boards can be constructed from regular bulletin boards using $\mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{Reg}}$ and standard techniques.

B UC Secure Public-Key Encryption and Constructions

It is well-known that the standard public-key encryption functionality $\mathcal{F}_{\mathsf{PKE}}$ from [9,12] can be UC-realized by any IND-CCA secure public-key encryption scheme.

Functionality $\mathcal{F}_{\mathsf{Sig}}$

Given an ideal adversary $\mathcal{S}$, verifiers $\mathcal{V}$ and a signer $\mathcal{P}_s$, $\mathcal{F}_{\mathsf{Sig}}$ performs:

Key Generation: Upon receiving a message $(\textsc{keygen}, sid)$ from $\mathcal{P}_s$, verify that $sid = (\mathcal{P}_s, sid')$ for some sid'. If not, ignore the request. Else, hand $(\textsc{keygen}, sid)$ to the adversary $\mathcal{S}$. Upon receiving $(\textsc{verification key}, sid, \mathsf{SIG}.vk)$ from $\mathcal{S}$, output $(\textsc{verification key}, sid, \mathsf{SIG}.vk)$ to $\mathcal{P}_s$, and record the pair $(\mathcal{P}_s, \mathsf{SIG}.vk)$.

Signature Generation: Upon receiving a message $(\textsc{sign}, sid, m)$ from $\mathcal{P}_s$, verify that $sid = (\mathcal{P}_s, sid')$ for some sid' . If not, then ignore the request. Else, send $(\textsc{sign}, sid, m)$ to $\mathcal{S}$. Upon receiving $(\textsc{signature}, sid, m, \sigma)$ from $\mathcal{S}$, verify that no entry $(m, \sigma, \mathsf{SIG}.vk, 0)$ is recorded. If it is, then output an error message to $\mathcal{P}_s$ and halt. Else, output $(\textsc{signature}, sid, m, \sigma)$ to $\mathcal{P}_s$, and record the entry $(m, \sigma, \mathsf{SIG}.vk, 1)$.

Signature Verification: Upon receiving a message $(\textsc{verify}, sid, m, \sigma, \mathsf{SIG}.vk')$ from some party $\mathcal{V}_i \in \mathcal{V}$, hand $(\textsc{verify}, sid, m, \sigma, \mathsf{SIG}.vk')$ to $\mathcal{S}$. Upon receiving $(\textsc{verified}, sid, m, \phi)$ from $\mathcal{S}$ do:

1. If $\mathsf{SIG}.vk' = \mathsf{SIG}.vk$ and the entry $(m, \sigma, \mathsf{SIG}.vk, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key $\mathsf{SIG}.vk'$ is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
2. Else, if $\mathsf{SIG}.vk' = \mathsf{SIG}.vk$, the signer $\mathcal{P}_s$ is not corrupted, and no entry $(m, \sigma', \mathsf{SIG}.vk, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, \sigma, \mathsf{SIG}.vk, 0)$. (This condition guarantees unforgeability: If $\mathsf{SIG}.vk'$ is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
3. Else, if there is an entry $(m, \sigma, \mathsf{SIG}.vk', f')$ recorded, then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
4. Else, let $f = \phi$ and record the entry $(m, \sigma, \mathsf{SIG}.vk', \phi)$.

Output $(\textsc{verified}, sid, m, f)$ to $\mathcal{V}_i$.

Fig. 9. Functionality $\mathcal{F}_{\mathsf{Sig}}$ for Digital Signatures.

One of the main building blocks of our framework is a UC-secure public-key encryption with a plaintext verification property formalized as functionality $\mathcal{F}_{\mathsf{PKEPV}}$ that is presented in Section 4.2. In order to realize $\mathcal{F}_{\mathsf{PKEPV}}$, we will show that it is possible to generate proofs that a given plaintext message was contained in a given ciphertext for the random oracle-based IND-CCA secure public-key encryption schemes of [25,16].

Semantics of a public-key encryption scheme. We consider public-key encryption schemes PKE that have public-key $\mathcal{PK}$, secret key $\mathcal{SK}$, message $\mathcal{M}$, randomness $\mathcal{R}$ and ciphertext $\mathcal{C}$ spaces that are functions of the security parameter κ, and consist of a PPT key generation algorithm KG, a PPT encryption algorithm Enc and a deterministic decryption algorithm Dec. For $(\mathsf{pk}, \mathsf{sk}) \overset{\$}{\leftarrow} \mathsf{KG}(1^\kappa)$, any

Functionality $\mathcal{F}_{\mathsf{Reg}}$

$\mathcal{F}_{\mathsf{Reg}}$ interacts with a set of parties $\mathcal{P}$ and an ideal adversary $\mathcal{S}$, proceeding as follows:

Key Registration: Upon receiving a message ($\textsc{register}, sid, \mathsf{pk}$) from a party $\mathcal{P}_i \in \mathcal{P}$, send ($\textsc{Registering}, sid, \mathsf{pk}$) to $\mathcal{S}$. Upon receiving (sid, ok) from $\mathcal{S}$, and if this is the first message from $\mathcal{P}_i$, then record the pair $(\mathcal{P}_i, \mathsf{pk})$.

Key Retrieval: Upon receiving a message ($\textsc{Retrieve}, sid, \mathcal{P}_j$) from a party $\mathcal{P}_i \in \mathcal{P}$, send message ($\textsc{Retrieve}, sid, \mathcal{P}_j$) to $\mathcal{S}$ and wait for it to return a message ($\textsc{Retrieve}$, sid, ok). Then, if there is a recorded pair $(\mathcal{P}_j, \mathsf{pk})$ output ($\textsc{Retrieve}, sid, \mathcal{P}_j, \mathsf{pk}$) to $\mathcal{P}_i$. Otherwise, if there is no recorded tuple, return ($\textsc{Retrieve}, sid, \mathcal{P}_j, \perp$).

Fig. 10. Functionality $\mathcal{F}_{\mathsf{Reg}}$ for Key Registration.

Functionality $\mathcal{F}_{\mathsf{BB}}$

$\mathcal{F}_{\mathsf{BB}}$ interacts with a set of parties $\mathcal{P}$ and an ideal adversary $\mathcal{S}$, proceeding as follows:

Register: Upon receiving ($\textsc{Init}, sid, \mathcal{P}$) by all parties in a set $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ where sid was not used before, store $\mathcal{P}$ locally.

Write: Upon receiving ($\textsc{Write}, sid, ssid, \mathcal{P}, m$) from a party $\mathcal{P}_i \in \mathcal{P}$, where $ssid$ was not used before for this sid, store the message m as $(sid, ssid, i, m)$.

Read: Upon receiving ($\textsc{Read}, sid$) from any party (possibly outside $\mathcal{P}$), the functionality returns all $(sid, ssid, i, m)$ that were stored.

Fig. 11. Functionality $\mathcal{F}_{\mathsf{BB}}$ for an authenticated Bulletin Board.

$\mathsf{m} \in \mathcal{M}$, and $\mathsf{ct} \overset{\$}{\leftarrow} \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$, it should hold that $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) = \mathsf{m}$ with overwhelming probability over the used randomness. Moreover, we consider public-key encryption schemes that are IND-CCA secure according to the definition considered in [9,12].

The Pointcheval [25] IND-CCA secure Cryptosystem. This cryptosystem can be constructed from any Partially Trapdoor One-Way Injective Function in the random oracle model. First we recall the definition of Partially Trapdoor One-Way Functions. As observed in [25], the classical El Gamal cryptosystem is a partially trapdoor one-way injective function under the Computational Diffie Hellman (CDH) assumption, implying an instantiation of this cryptosystem under CDH.

Definition 8 (Partially Trapdoor One-Way Function [25]). *The function $f : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$ is said to be partially trapdoor one-way if:*

- *For any given $z = f(x, y)$, it is computationally impossible to get back a compatible x. Such an x is called a partial preimage of z. More formally, for any polynomial time adversary A, its success, defined by $\mathtt{Succ}_{\mathtt{A}} = Pr_{x,y}[\exists y'$,*

$f(x', y') = f(x, y) | x' = A(f(x, y))]$, is negligible. It is one-way even for just finding partial-preimage, thus partial one-wayness.

– *Using some extra information (the trapdoor), for any given $z \in f(\mathcal{X} \times \mathcal{Y})$, it is easily possible to get back an x, such that there exists a y which satisfies $f(x, y) = z$. The trapdoor does not allow a total inversion, but just a partial one and it is thus called a partial trapdoor.*

Let's now recall the construction of [25], which is presented in Definition 9.

Definition 9 (Pointcheval [25] IND-CCA secure Cryptosystem). *Let $\mathcal{TD}$ be a family of partially trapdoor one-way injective functions and let $H : \{0,1\}^{|m|+\kappa} \to \mathcal{Y}$ and $G : \mathcal{X} \to \{0,1\}^{|m|+\kappa}$ be random oracles, where $|m|$ is message length. This cryptosystem consists of a triple of algorithms $\mathsf{PKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ that work as follows:*

– $\mathsf{KG}(1^\kappa)$*: Sample a random partially trapdoor one-way injective function $f : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$ from $\mathcal{TD}$ and denote its inverse parameterized by the trapdoor by $f^{-1} : \mathcal{Z} \to \mathcal{X}$. The public-key is $\mathsf{pk} = f$ and the secret key is $\mathsf{sk} = (f, f^{-1})$.*
– $\mathsf{Enc}(\mathsf{pk}, \mathsf{m})$*: Sample $r \xleftarrow{\$} \mathcal{X}$ and $s \xleftarrow{\$} \{0,1\}^\kappa$. Compute $a \leftarrow f(r, H(\mathsf{m}||s))$ and $b = (\mathsf{m}||s) \oplus G(r)$, outputting $\mathsf{ct} = (a, b)$ as the ciphertext.*
– $\mathsf{Dec}\ (\mathsf{sk}, \mathsf{ct})$*: Given a ciphertext $\mathsf{ct} = (a, b)$ and secret key $\mathsf{sk} = f^{-1}$, compute $r \leftarrow f^{-1}(a)$ and $M \leftarrow b \oplus G(r)$. If $a = f(r, H(M))$, parse $M = (m||s)$ and output m. Otherwise, output $\perp$.*

Properties of the Pointcheval [25] IND-CCA secure Cryptosystem. First, notice that this construction can be instantiated in the restricted observable and programmable global random oracle model of [6]. Next, we observe that this construction is witness recovering, meaning that it allows for the decrypting party to recover all of the randomness used in generating a ciphertext (*i.e.* r and s). Moreover, this construction is committing, meaning that it is infeasible for an adversary to obtain two pairs of messages and randomness that result in the same ciphertext. We now recall the definitions of witness recovering and committing encryption schemes.

Definition 10 (Witness-Recovering Public-Key Encryption). *A public-key encryption scheme is witness-recovering if its decryption algorithm Dec takes as input a secret key $\mathsf{sk} \in \mathcal{SK}$ and a ciphertext $\mathsf{ct} \in \mathcal{C}$ and outputs either a pair (m, r) for $\mathsf{m} \in \mathcal{M}$ and $\mathsf{r} \in \mathcal{R}$ or an error symbol $\perp$. For any $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KG}(1^\kappa)$, any $\mathsf{m} \in \mathcal{M}$, any $\mathsf{r} \xleftarrow{\$} \mathcal{R}$ and $c \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m}; \mathsf{r})$, it holds that $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) = (\mathsf{m}, \mathsf{r})$ with overwhelming probability over the randomness used by the algorithms.*

Definition 11 (Committing Encryption). *Let PKE be a public-key encryption scheme and κ a security parameter. For every PPT adversary $\mathcal{A}$, it holds that:*

$$\Pr\left[\mathsf{Enc}(\mathsf{pk},\mathsf{m}_0;\mathsf{r}_0)=\mathsf{Enc}(\mathsf{pk},\mathsf{m}_1;\mathsf{r}_1)\;\middle|\;\begin{array}{c}\mathsf{pk}\xleftarrow{\$}\mathcal{PK},\\(\mathsf{r}_0,\mathsf{r}_1,\mathsf{m}_0,\mathsf{m}_1)\xleftarrow{\$}\mathcal{A}(\mathsf{pk}),\\\mathsf{r}_0,\mathsf{r}_1\in\mathcal{R},\mathsf{m}_0,\mathsf{m}_1\in\mathcal{M},\\\mathsf{m}_0\neq\mathsf{m}_1\end{array}\right]\in\mathsf{negl}(\kappa)$$

The Pointcheval [25] IND-CCA secure Cryptosystem is trivially witness-recovering since a decrypting party always recovers the randomness (r, s) used for generating a ciphertext. In order to see why it is also committing, notice that an adversary can only make a polynomial number of queries to $H(\cdot)$, so it can only find a pair (m', s') such that $(m', s') \neq (m, s)$ and $H(m||s) = H(m'||s')$ with negligible probability. Analogously, the adversary can only find an r' such that $r' \neq r$ and $G(r) = G(r')$ with negligible probability. Hence, since f is injective, the adversary can only find (m', s', r') such that $(m', s', r') \neq (m, s, r)$ and $f(r', H(m'||s')) = f(r, H(m||s))$ with negligible probability.

Plaintext Verification for the Pointcheval [25] IND-CCA secure Cryptosystem. We first extend the semantics of public-key encryption by adding a plaintext verification algorithm $\{0,1\} \leftarrow \mathsf{V}(\mathsf{ct},\mathsf{m},\pi)$ that outputs 1 if m is the plaintext message contained in ciphertext ct given a valid proof π that also contains the public-key pk used to generate the ciphertext. Furthermore, we modify the encryption and decryption algorithms as follows: $(\mathsf{ct},\pi) \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk},\mathsf{m})$ and $(\mathsf{m},\pi) \leftarrow \mathsf{Dec}(\mathsf{sk},\mathsf{ct})$ now output a valid proof π that m is contained in ct. The security guarantees provided by the verification algorithm are laid out in Definition 12. Notice that this definition only considers cryptosystems where the proof π consists of the randomness used by the encryption algorithm, which is enough for our version of the Pointcheval [25] IND-CCA secure cryptosystem with plaintext verification. A generalization of this definition follows by defining the space of plaintext validity proofs and requiring that π, π' are in that space, as well as that the adversary provides ct, since it might not be computable from (m, π).

Definition 12 (Plaintext Verification). *Let* $\mathsf{PKE} = (\mathsf{KG},\mathsf{Enc},\mathsf{Dec},\mathsf{V})$ *be a public-key encryption scheme and* κ *be a security parameter. For every PPT adversary* $\mathcal{A}$*, it holds that:*

$$\Pr\left[\mathsf{V}(\mathsf{ct},\mathsf{m}',\pi')=1\;\middle|\;\begin{array}{c}\mathsf{pk}\xleftarrow{\$}\mathcal{PK},\\(\mathsf{m},\pi,\mathsf{m}',\pi')\xleftarrow{\$}\mathcal{A}(\mathsf{pk}),\\\pi=(\mathsf{pk},r),\pi'=(\mathsf{pk},r')\in\mathcal{PK}\cup\mathcal{R},\\\mathsf{m},\mathsf{m}'\in\mathcal{M},(\mathsf{ct},\pi)\leftarrow\mathsf{Enc}(\mathsf{pk},\mathsf{m};r),\mathsf{m}'\neq\mathsf{m}\end{array}\right]\in\mathsf{negl}(\kappa)$$

We can extended the Pointcheval [25] IND-CCA secure cryptosystem to add plaintext verification as follows:

Definition 13 (Pointcheval [25] IND-CCA Secure Cryptosystem with Plaintext Verification). *Let* $\mathcal{TD}$ *be a family of partially trapdoor one-way* *injective functions and let* $H : \{0,1\}^{|m|+\kappa} \to \mathcal{Y}$ *and* $G : \mathcal{X} \to \{0,1\}^{|m|+\kappa}$ *be random oracles, where* $|m|$ *is message length. This cryptosystem consists of the algorithms* $\mathsf{PKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}\,\mathsf{V})$ *that work as follows:*

– $\mathsf{KG}(1^\kappa)$*: Same as in Definition 9.*
– $\mathsf{Enc}(\mathsf{pk}, \mathsf{m})$*: Same as in Definition 9 but also output a proof* $\pi = (\mathsf{pk}, r, s)$ *(i.e. the encryption randomness) besides the ciphertext* $\mathsf{ct} = (a, b)$*.*
– $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$*: Same as in Definition 9 but also output a proof* $\pi = (\mathsf{pk}, r, s)$ *(i.e. the retrieved encryption randomness) besides the plaintext message* m*.*
– $\mathsf{V}(\mathsf{ct}, \mathsf{m}, \pi)$*: Parse* $\pi = (\mathsf{pk}, r, s)$*, compute* $\mathsf{ct}' \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m}, (r, s))$ *and output* 1 *if and only if* $\mathsf{ct} = \mathsf{ct}'$*.*

Using the facts that this cryptosystem is witness-recovering and committing, both the encrypting and decrypting parties can generate a proof $\pi = (\mathsf{pk}, r, s)$ that a message m was encrypted under public-key pk with randomness (r, s) resulting in ciphertext ct. Notice that the witness-recovering property ensures that a decrypting party is able to recover the randomness (r, s) too. Any third party verifier with input $(\mathsf{ct}, \mathsf{m}, \pi)$ can execute the verification algorithm $\mathsf{V}(\mathsf{ct}, \mathsf{m}, \pi)$ and obtain 1 if and only if π is a valid proof that m is contained in ct. Notice that an adversary cannot present two different triples (m, s, r) and (m', s', r') that pass this test with the same public-key pk except with negligible probability, since the cryptosystem is committing as discussed above. Assuming by contradiction that such an adversary $\mathcal{A}$ exists, we can construct an adversary $\mathcal{A}'$ that wins the game of Definition 11 with non-negligible probability. Adversary $\mathcal{A}'$ receives pk from the challenger in the game of Definition 11 and then acts as the challenger in the game of Definition 12, relaying pk to $\mathcal{A}$. Upon receiving $(\mathsf{m}, \pi, \mathsf{m}', \pi')$ from $\mathcal{A}$, it relays $(\mathsf{m}, \pi, \mathsf{m}', \pi')$ to the the challenger in the game of Definition 11 as $(\mathsf{r}_0, \mathsf{r}_1, \mathsf{m}_0, \mathsf{m}_1)$. Notice that, for the extended cryptosystem above, $1 \leftarrow \mathsf{V}(\mathsf{ct}, \mathsf{m}, \pi)$ occurs if and only if $\mathsf{ct} = \mathsf{ct}'$, where $\mathsf{ct}' \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m}; (r, s))$ and $\pi = (\mathsf{pk}, r, s)$. This implies that, if adversary $\mathcal{A}$ wins the game of Definition 12 with non-negligible probability, it is able to produce two messages m, m' and corresponding proofs $\pi = (\mathsf{pk}, r, s), \pi' = (\mathsf{pk}', r', s')$ for which $\mathsf{m} \neq \mathsf{m}'$ and $\mathsf{Enc}(\mathsf{pk}, \mathsf{m}; \pi = (r, s)) = \mathsf{Enc}(\mathsf{pk}, \mathsf{m}'; \pi' = (r', s'))$ with non-negligible probability. Hence, adversary $\mathcal{A}'$ wins the game of Definition 11 with non-negligible probability.

C Realizing the Secret Joint Authentication Functionality

C.1 Realizing $\mathcal{F}_{\mathsf{PSAuth}}$

The basic idea for realizing $\mathcal{F}_{\mathsf{PSAuth}}$ is using a signature scheme (captured by $\mathcal{F}_{\mathsf{Sig}}$) and a certified encryption scheme with plaintext verification (captured by $\mathcal{F}_{\mathsf{CPKEPV}}$), *i.e.* an encryption scheme with two crucial properties: 1. An encrypting party is guaranteed to encrypt a message that can only be opened by the intended receiver (*i.e.* it is possible to make sure the public-key used belongs to the intended receiver of the encrypted messages); 2. Both encrypting and decrypting parties can generate publicly verifiable proofs that a certain message was contained in a given ciphertext. The private channel itself is realized by encrypting messages under the encryption scheme, while joint authentication is achieved by having all parties in $\mathcal{P}$ (including the sender) and bureaucrats in $\mathcal{B}$ sign the resulting ciphertext. In order to obtain efficiency, a joint public/secret key pair is generated for each set of receivers, in such a way that the same ciphertext can be decrypted by all the receivers holding the corresponding joint secret key. Later on, if any party in $\mathcal{P}$ (including the sender) wishes to start the verification procedure to prove that a certain message was indeed contained in the ciphertext associated with a given *ssid*, it recovers the plaintext message and a proof of plaintext validity from the ciphertext and sends those to one or more verifiers. With these values, any party can first verify that the ciphertext that was sent indeed corresponds to that message due to the plaintext verification property of the encryption scheme and then verify that it has been jointly authenticated by checking that there exist valid signatures on that ciphertext by all parties in $\mathcal{P}$ and bureaucrats in $\mathcal{B}$.

In order to obtain $\mathcal{F}_{\mathsf{CPKEPV}}$, we first define and realize an ideal functionality for public-key encryption with plaintext verification $\mathcal{F}_{\mathsf{PKEPV}}$. This functionality and the protocol that realizes it are extensions of the results of [9,14], which show that IND-CCA secure encryption schemes UC realize the standard public-key encryption functionality. In our definition and construction, we show that IND-CCA encryption scheme with an additional plaintext verification property (*e.g.* as the scheme discussed in Section B) UC realize $\mathcal{F}_{\mathsf{PKEPV}}$. Building on $\mathcal{F}_{\mathsf{PKEPV}}$ and a key registration functionality $\mathcal{F}_{\mathsf{Reg}}$, we define and realize $\mathcal{F}_{\mathsf{CPKEPV}}$. We again extend a functionality and protocol from [12], which shows that certified public-key encryption can be realized from the standard public-key encryption functionality and $\mathcal{F}_{\mathsf{Reg}}$. Following a similar approach, we show a protocol based on $\mathcal{F}_{\mathsf{PKEPV}}$ and $\mathcal{F}_{\mathsf{Reg}}$ that UC realizes $\mathcal{F}_{\mathsf{CPKEPV}}$.

Public-Key Encryption with Plaintext Verification $\mathcal{F}_{\mathsf{PKEPV}}$ We will use a public-key encryption scheme that allows for both the party generating a ciphertext and the party decrypting it to obtain a publicly verifiable proof that a given message was contained in such ciphertext. Notice that this is not a zero-knowledge proof, but a proof whose verification requires the message to be revealed. We model such an encryption scheme by functionality $\mathcal{F}_{\mathsf{PKEPV}}$, which is an extension of the standard public-key encryption functionality $\mathcal{F}_{\mathsf{PKE}}$ from [9,14] with a new plaintext verification interface for verifying that a given plaintext was contained in a given ciphertext. This plaintext verification interface is incorporated into the functionality following the same approach as in [9,14]: the functionality first looks up the corresponding ciphertext and message pair on an internal list (*i.e.* where it should be in case the ciphertext was generated by the functionality), returning 1 is such a pair exists; otherwise, if the ciphertext is not contained in this internal list (*i.e.* it has been generated by an adversary in a potentially incorrect way), the functionality performs the verification procedure internally, by attempting to decrypt the ciphertext and then executing the verification algorithm taking as input the ciphertext along with the resulting plaintext message and proof, returning the output of this algorithm to the verifier. It is well-known that IND-CCA secure public-key encryption schemes can be used to realize $\mathcal{F}_{\mathsf{PKE}}$ as defined in [9,14], but we will show that there exist IND-CCA secure public-key encryption schemes [25,16] that also realize our extended functionality $\mathcal{F}_{\mathsf{PKEPV}}$.

Realizing $\mathcal{F}_{\mathsf{PKEPV}}$ It is known [9,14] that an IND-CCA secure public encryption scheme realizes the key generation, encryption and decryption interfaces of $\mathcal{F}_{\mathsf{PKEPV}}$ (without generating proofs), which correspond to the standard public-key encryption functionality $\mathcal{F}_{\mathsf{PKE}}$ from [9,14]. The missing pieces in realizing our formulation of $\mathcal{F}_{\mathsf{PKEPV}}$ are algorithms for generating and verifying proofs that a given plaintext is contained in a given ciphertext produced by a IND-CCA secure public encryption scheme. Notice that these proofs need not to be zero-knowledge, as they can be verified given the plaintext message and the corresponding ciphertext. We use the version of Pointcheval's IND-CCA secure cryptosystem [25] with plaintext verification from Section B to realize $\mathcal{F}_{\mathsf{PKEPV}}$ following the same approach as in [9,14]. This generic construction works in the restricted programmable and observable random oracle model [6] and can be instantiated from the CDH assumption.

We realize $\mathcal{F}_{\mathsf{PKEPV}}$ by extending the encryption protocol π_{PKE} of [9,14], which is constructed from any IND-CCA secure cryptosystem $\mathsf{PKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$. We obtain a protocol π_{PKEPV} that realizes $\mathcal{F}_{\mathsf{PKEPV}}$ based on an IND-CCA public-key encryption scheme with plaintext verification $\mathsf{PKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{V})$ as defined in Section B. Protocol π_{PKEPV} works as follows: Upon receiving ($\textsc{KeyGen}$, $sid, \mathcal{P}_{owner}$), $\mathcal{P}_{owner}$ executes $(\mathsf{sk}, \mathsf{pk}) \overset{\$}{\leftarrow} \mathsf{KG}(1^{\kappa})$, records sk and returns pk. Upon receiving a message ($\textsc{Encrypt}, sid, \mathcal{P}_{owner}, \mathbf{e}', m$), any party $\mathcal{P}_i \in \mathcal{P}$ outputs ct where $(\mathsf{ct}, \pi) \overset{\$}{\leftarrow} \mathsf{Enc}(\mathbf{e}', m)$ if $m \in M$ (otherwise it outputs an error message). Upon receiving ($\textsc{Decrypt}, sid, \mathcal{P}_{owner}, c$), $\mathcal{P}_{owner}$ outputs m where $(m, \pi) \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$. Upon receiving a message ($\textsc{Verify}, sid, \mathcal{P}_{owner}, c, m, \pi$), a verifier $\mathcal{V}_i \in \mathcal{V}$ outputs b where $b \leftarrow \mathsf{V}(c, m, \pi)$.

We will prove that the public-key encryption scheme with plaintext verification of Definition 13 can be used to instantiate Π_{PKEPV} in such a way that it realizes $\mathcal{F}_{\mathsf{PKEPV}}$. We leave a more general proof as a future work.

Theorem 2. *Let* $\mathsf{PKE} = \{\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{V}\}$ *be the public-key encryption scheme with plaintext verification of Definition 13. Protocol* Π_{PKEPV} *instantiated with* PKE *UC realizes* $\mathcal{F}_{\mathsf{PKEPV}}$ *in the restricted programmable and observable random oracle model [6].*

Proof. In order to prove this construction securely realizes $\mathcal{F}_{\mathsf{PKEPV}}$, we construct a simulator such that no environment can distinguish an ideal execution with this simulator and $\mathcal{F}_{\mathsf{PKEPV}}$ from a real execution of Π_{PKEPV} with any adversary $\mathcal{A}$ and dummy parties. Notice that the steps of Π_{PKEPV} dealing with messages ($\textsc{KeyGen}, sid, \mathcal{P}_{owner}$), ($\textsc{Encrypt}, sid, \mathcal{P}_{owner}, \mathbf{e}', m$) and ($\textsc{Decrypt}$,

Functionality $\mathcal{F}_{\mathsf{PKEPV}}$

$\mathcal{F}_{\mathsf{PKEPV}}$ interacts with a special decrypting party $\mathcal{P}_{owner}$, a set of parties $\mathcal{P}$, a set of public verifiers $\mathcal{V}$ and an ideal adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{PKEPV}}$ is parameterized by a message domain ensemble $M = \{M_k\}_{k \in \mathcal{N}}$, a family of formal encryption algorithms $\{E_e\}_e$, a family of formal decryption algorithms $\{D_d\}_d$ for unregistered ciphertexts and a family of formal plaintext verification algorithms $\{V_v\}_v$. $\mathcal{F}_{\mathsf{PKEPV}}$ proceeds as follows:

Key Generation: Upon receiving a message $(\textsc{KeyGen}, sid, \mathcal{P}_{owner})$ from a party $\mathcal{P}_{owner} \in \mathcal{P}$ (or $\mathcal{S}$), proceed as follows:

1. Send $(\textsc{KeyGen}, sid, \mathcal{P}_{owner})$ to $\mathcal{S}$.
2. Receive a value $\mathbf{e}$ from $\mathcal{S}$.
3. Record $\mathbf{e}$ and output $\mathbf{e}$ to $\mathcal{P}_{owner}$.

Encryption: Upon receiving a message $(\textsc{Encrypt}, sid, \mathcal{P}_{owner}, \mathbf{e}', m)$ from a party $\mathcal{P}_i \in \mathcal{P}$, proceed as follows:

1. If $m \notin M$, then return an error message to $\mathcal{P}_i$.
2. If $m \in M$, then:
   - If $\mathcal{P}_{owner}$ is corrupted, or $\mathbf{e}' \neq \mathbf{e}$, then compute $(c, \pi) \leftarrow E_k(m)$.
   - Otherwise, let $(c, \pi) \leftarrow E_k(1^{|m|})$.

   Record the pair (m, c, π) and return (c, π) to $\mathcal{P}_i$.

Decryption: Upon receiving a message $(\textsc{Decrypt}, sid, \mathcal{P}_{owner}, c)$ from $\mathcal{P}_{owner}$, proceed as follows (if the input is from another party then ignore):

1. If there is a recorded tuple (c, m, π), then hand (m, π) to $\mathcal{P}_{owner}$. (If there is more than one value m that corresponds to c then unique decryption is not possible. In that case, output an error message to $\mathcal{P}_{owner}$).
2. Otherwise, compute $(m, \pi) \leftarrow D(c)$ and hand (m, π) to $\mathcal{P}_{owner}$.

Plaintext Verification: Upon receiving a message $(\textsc{Verify}, sid, \mathcal{P}_{owner}, c, m, \pi)$ from a verifier $\mathcal{V}_i \in \mathcal{V}$, proceed as follows:

1. If there is a recorded tuple (c, m, π), then output 1 to $\mathcal{V}_i$.
2. Otherwise, compute $b \leftarrow V(c, m, \pi)$, outputting b to $\mathcal{V}_i$.

Fig. 12. Public-Key Encryption Functionality with Plaintext Verification $\mathcal{F}_{\mathsf{PKEPV}}$.

$sid, \mathcal{P}_{owner}, c)$ correspond exactly to the protocol of [9,14] realizing the standard public-key encryption, and our simulator can function exactly as the simulator of [9,14]. In fact, all the simulator does is executing $\mathsf{KG}(1^\kappa)$ and setting pk. It is proven in [9,14] that such a simulator results in an execution indeed indistinguishable from the real protocol execution with an adversary $\mathcal{A}$ and the same argument can be used in our case. As for the remaining message $(\textsc{Verify}, sid, \mathcal{P}_{owner}, c, m, \pi)$, any party in the simulation will output exactly the same as in the real protocol, since the output will either come from the simulator if it indeed simulated a ciphertext generation for m that resulted in (c, m, π) (meaning the ciphertext was correctly/honestly generated) or whatever the output of $\mathsf{V}(c, m, \pi)$ is (in case the ciphertext was not generated by the simulated functionality). Special care needs to be taken when simulating the verification of a ciphertext simulated for an honest party, which is computed as $\mathsf{Enc}(\mathsf{pk}, 1; \mathsf{r})$

(for a random r) instead of using the actual message given by the honest party. In this case, when π is revealed, it is incompatible with the message 1 in the ciphertext. However, upon receiving the actual message m the simulator shows the adversary answers to queries $H(\mathsf{m}\|s)$ and $G(r)$ that match m and ct.

Functionality $\mathcal{F}_{\mathsf{CPKEPV}}$

$\mathcal{F}_{\mathsf{CPKEPV}}$ interacts with a special decrypting party $\mathcal{P}_{owner}$, a set of parties $\mathcal{P}$, a set of public verifiers $\mathcal{V}$ and an ideal adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{PKEPV}}$ is parameterized by a message domain ensemble $M = \{M_k\}_{k \in \mathcal{N}}$, a family of formal encryption algorithms $\{E_e\}_e$, a family of formal decryption algorithms $\{D_d\}_d$ for unregistered ciphertexts a family of formal plaintext verification algorithms $\{V_v\}_v$. $\mathcal{F}_{\mathsf{CPKEPV}}$ proceeds as follows:

Encryption: Upon receiving a message $(\textsc{Encrypt}, sid, \mathcal{P}_{owner}, m)$ from a party $\mathcal{P}_i \in \mathcal{P}$, proceed as follows:

1. if this is the first encryption request made by $\mathcal{P}_i$ then notify $\mathcal{S}$ that π made an encryption request.
2. If $m \notin M$, then return an error message to $\mathcal{P}_1$.
3. If $m \in M$, then:
   - If $\mathcal{P}_{owner}$ is corrupted, then compute $(c, \pi) \leftarrow E_k(m)$.
   - Otherwise, let $(c, \pi) \leftarrow E_k(1^{|m|})$.

   Record the pair (m, c, π) and return (c, π) to $\mathcal{P}_i$.

Decryption: Upon receiving a message $(\textsc{Decrypt}, sid, \mathcal{P}_{owner}, c)$ from $\mathcal{P}_{owner}$, proceed as follows (if the input is from another party then ignore):

1. If this is the first decryption request made by $\mathcal{P}_{owner}$ then notify $\mathcal{S}$ that a decryption request was made.
2. If there is a recorded tuple (c, m, π), then hand m, π to $\mathcal{P}_{owner}$. (If there is more than one value m that corresponds to c then unique decryption is not possible. In that case, output an error message to $\mathcal{P}_{owner}$).
3. Otherwise, compute $(m, \pi) \leftarrow D(c)$ and hand (m, π) to $\mathcal{P}_{owner}$.

Plaintext Verification: Upon receiving a message $(\textsc{Verify}, sid, \mathcal{P}_{owner}, c, m, \pi)$ from a verifier $\mathcal{V}_i \in \mathcal{V}$ output 1 to $\mathcal{V}_i$ if there is a recorded tuple (c, m, π). Otherwise, output 0.

Fig. 13. Certified Public-Key Encryption Functionality with Plaintext Verification $\mathcal{F}_{\mathsf{CPKEPV}}$.

Certified Encryption With Plaintext Verification $\mathcal{F}_{\mathsf{CPKEPV}}$ We are now ready to define and construct a version of certified public-key encryption with plaintext verification following the approach of [12]. Essentially, certified public-key encryption captures a notion where public-keys are not explicitly available but are linked to specific parties, guaranteeing that an encrypted message will

Protocol π_{CPKEPV}

π_{CPKEPV} is parameterized by the families $\{E_e\}_e$, $\{D_d\}_d$ and $\{V_v\}_v$ of algorithms of the functionality it is to realize. A special decrypting party $\mathcal{P}_{owner}$, a set of parties $\mathcal{P}$, a set of public verifiers $\mathcal{V}$ execute π_{CPKEPV} as follows:

Initialization: At the first activation an instance of $\mathcal{F}_{\mathsf{PKEPV}}$ is instantiated with the families $\{E_e\}_e$, $\{D_d\}_d$ and $\{V_v\}_v$. Party $\mathcal{P}_{owner}$ sends message (KeyGen, $sid, \mathcal{P}_{owner}$) to $\mathcal{F}_{\mathsf{PKEPV}}$, receiving pk. Next, $\mathcal{P}_{owner}$ sends (register, sid, pk) to $\mathcal{F}_{\mathsf{Reg}}$.

Encryption: Upon receiving a message (Encrypt, $sid, \mathcal{P}_{owner}, \mathbf{e}', m$), party $\mathcal{P}_i \in \mathcal{P}$ proceed as follows:

1. Check whether it has a recorded public-key $\mathbf{e}$. If not, send (Retrieve, $sid, \mathcal{P}_{owner}$) to $\mathcal{F}_{\mathsf{Reg}}$, receiving (Retrieve, $sid, \mathcal{P}_{owner}, \mathsf{pk}$) as response. If $\mathsf{pk} \neq \perp$, record $\mathbf{e} = \mathsf{pk}$. Otherwise, return $\perp$.
2. If $\mathbf{e} \neq \perp$, send (Encrypt, $sid, \mathcal{P}_{owner}, \mathbf{e}', m$) to $\mathcal{F}_{\mathsf{PKEPV}}$, receiving (c, π) as response. Output c and record the tuple (m, c, π).

Decryption: Upon receiving a message (Decrypt, $sid, \mathcal{P}_{owner}, c$), $\mathcal{P}_{owner}$ sends a message (Decrypt, $sid, \mathcal{P}_{owner}, c$) to $\mathcal{F}_{\mathsf{PKEPV}}$, receiving and outputting (m, π).

Plaintext Verification: Upon receiving a message (Verify, $sid, \mathcal{P}_{owner}, c, m, \pi$), a verifier $\mathcal{V}_i \in \mathcal{V}$ proceeds as follows:

1. Check whether it has a recorded public-key $\mathbf{e}$. If not, send (Retrieve, $sid, \mathcal{P}_{owner}$) to $\mathcal{F}_{\mathsf{Reg}}$, receiving (Retrieve, $sid, \mathcal{P}_{owner}, \mathsf{pk}$) as response. If $\mathsf{pk} \neq \perp$, record $\mathbf{e} = \mathsf{pk}$. Otherwise, return 0.
2. Obtain pk from π. If $\mathsf{pk} = \mathbf{e}$, compute $b \leftarrow V(c, m, \pi)$ and outputs b. Otherwise, output 0.

Fig. 14. Protocol π_{CPKEPV} realizing $\mathcal{F}_{\mathsf{CPKEPV}}$.

be received by an specific party. In order to realize such a functionality, a key registration ideal functionality $\mathcal{F}_{\mathsf{Reg}}$ that allows parties to register their public-keys is required. It was shown in [12] that certified public-key encryption can be realized from a standard public encryption functionality and $\mathcal{F}_{\mathsf{Reg}}$. We will extend both the original functionality and protocol from [12] to incorporate plaintext verification, showing that $\mathcal{F}_{\mathsf{CPKEPV}}$ can be realized from $\mathcal{F}_{\mathsf{PKEPV}}$ and $\mathcal{F}_{\mathsf{Reg}}$. The notion of certified public-key encryption with plaintext verification is captured by functionality $\mathcal{F}_{\mathsf{CPKEPV}}$ introduced in Figure 13. Notice that the Plaintext Verification interface of $\mathcal{F}_{\mathsf{CPKEPV}}$ only outputs 1 if it receives a query with a tuple (c, m, π) that is registered in the functionality's internal list. This captures the fact that only ciphertexts generated by the functionality with a party's legitimate public-key (as encoded in the encryption algorithm $E_k(\cdot)$ are considered valid, while arbitrary ciphertexts or ciphertexts generated from other public-keys are automatically considered invalid.

Realizing $\mathcal{F}_{\mathsf{CPKEPV}}$ We follow the approach of [12] to realize $\mathcal{F}_{\mathsf{CPKEPV}}$ from a public-key encryption scheme with plaintext verification $\mathcal{F}_{\mathsf{PKEPV}}$ and a key registration functionality $\mathcal{F}_{\mathsf{Reg}}$. Our protocol implements Initialization, Encryption and Decryption interfaces exactly as in [12] and follows the same approach for implementing the Plaintext Verification interface. Protocol π_{CPKEPV} realizing $\mathcal{F}_{\mathsf{CPKEPV}}$ is presented in Figure 14.

Theorem 3. *Protocol π_{CPKEPV} UC realizes $\mathcal{F}_{\mathsf{PKEPV}}$ in the $(\mathcal{F}_{\mathsf{PKEPV}}, \mathcal{F}_{\mathsf{Reg}})$-hybrid model.*

Proof. In order to see why Protocol π_{CPKEPV} is secure, notice that a simulator $\mathcal{S}$ can be constructed exactly as in [12]: $\mathcal{S}$ runs with an internal copy of the adversary $\mathcal{A}$ towards which it simulates $\mathcal{F}_{\mathsf{PKEPV}}$ and $\mathcal{F}_{\mathsf{Reg}}$ exactly as described, simulating the process of registration by $\mathcal{P}_{owner}$ when $\mathcal{F}_{\mathsf{CPKEPV}}$ informs $\mathcal{S}$ that either encryption or decryption requests happened, as well as simulating the process of key retrieval when notified by the functionality. Notice that the ideal execution with the simulator and $\mathcal{F}_{\mathsf{CPKEPV}}$ is exactly the same as the real execution of Protocol π_{CPKEPV} with an adversary $\mathcal{A}$, as in the case of the protocol proposed in [12]. Hence, no environment can distinguish the ideal world simulation from the real world execution.

Secret Joint Authentication Protocol We can now construct a protocol π_{PSAuth} that realizes $\mathcal{F}_{\mathsf{PSAuth}}$ from $\mathcal{F}_{\mathsf{CPKEPV}}$, $\mathcal{F}_{\mathsf{Sig}}$ and $\mathcal{F}_{\mathsf{Reg}}$. This protocols starts by initializing an instance of $\mathcal{F}_{\mathsf{CPKEPV}}$ that is jointly used by all parties $\mathcal{P}_i \in \mathcal{P}$ (*i.e.* all parties in $\mathcal{P}$ act as $\mathcal{P}_{owner}$) and initializing an instance of $\mathcal{F}_{\mathsf{Sig}}$ for $\mathcal{P}_{sender}$, each party $\mathcal{P}_i \in \mathcal{P}$ and each bureaucrat $\mathcal{B}_i \in \mathcal{B}$. Next, $\mathcal{P}_{sender}$, parties in $\mathcal{P}$ and bureaucrats generate a signature verification key from their instances of $\mathcal{F}_{\mathsf{Sig}}$ and register it with $\mathcal{F}_{\mathsf{Reg}}$. When $\mathcal{P}_{sender}$ wants to send a message, it encrypts it using $\mathcal{F}_{\mathsf{CPKEPV}}$, signs the resulting ciphertext using $\mathcal{F}_{\mathsf{Sig}}$ and sends the resulting signature along with the ciphertext to all other parties and bureaucrats. All parties in $\mathcal{P}$ and all bureaucrats retrieve $\mathcal{P}_{sender}$'s key from $\mathcal{F}_{\mathsf{Reg}}$ and issue a verification query to $\mathcal{F}_{\mathsf{Sig}}$ to check that the signature on the ciphertext is valid. If this is the case, each bureaucrat uses its instance of $\mathcal{F}_{\mathsf{Sig}}$ to compute a signature on the ciphertext, which it sends to all other parties. Additionally, each party $\mathcal{P}_i \in \mathcal{P}$ decrypts the ciphertext using $\mathcal{F}_{\mathsf{CPKEPV}}$, obtaining a plaintext message and proof of plaintext validity (which it verifies using $\mathcal{F}_{\mathsf{CPKEPV}}$). In case both decryption and signature checks succeed, each party $\mathcal{P}_i$ computes a signature on the ciphertext using $\mathcal{F}_{\mathsf{Sig}}$ and sends it to all other parties and bureaucrats. In case either $\mathcal{P}_{sender}$ or a party $\mathcal{P}_i \in \mathcal{P}$ want to prove a certain message was sent by $\mathcal{P}_{sender}$ and jointly authenticated, it reveals the ciphertext, the message and proof of plaintext validity obtained by decrypting the ciphertext along with all signatures on that ciphertext (by $\mathcal{P}_{sender}$, all parties $\mathcal{P}_i \in \mathcal{P}$ and all bureaucrats) to a verifier, who can retrieve all signature verification keys from $\mathcal{F}_{\mathsf{Reg}}$, verify all signatures using $\mathcal{F}_{\mathsf{Sig}}$ and finally use the ciphertext, message and proof of plaintext validity to verify the plaintext with $\mathcal{F}_{\mathsf{CPKEPV}}$. Protocol π_{PSAuth} is described in Figures 15, 16.

Theorem 4. *Protocol π_{PSAuth} UC realizes $\mathcal{F}_{\mathsf{PSAuth}}$ in the $(\mathcal{F}_{\mathsf{CPKEPV}}, \mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{Reg}})$-hybrid model.*

Protocol π_{PSAuth}

π_{PSAuth} is parameterized by a special party $\mathcal{P}_{sender}$, a set of authenticating parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$, a set of bureaucrats $\mathcal{B} = \{\mathcal{B}_1, \dots, \mathcal{B}_b\}$ and a set of public verifiers $\mathcal{V}$ (s.t. $\mathcal{B} \subset \mathcal{V}$). π_{PSAuth} proceeds as follows:

Initialization: At the first activation, an instance of $\mathcal{F}_{\mathsf{CPKEPV}}$ is instantiated with the families $\{E_e\}_e$, $\{D_d\}_d$ and $\{V_v\}_v$ and all parties $\mathcal{P}_i \in \pi$ acting as $\mathcal{P}_{owner}$ (*e.g.* $\mathcal{P}_{owner} = \mathcal{P}$[a]. For each party in $\mathcal{P}$, bureaucrat in $\mathcal{B}$ and party $\mathcal{P}_{sender}$, an instance of $\mathcal{F}_{\mathsf{Sig}}$ is initialized with that party acting as $\mathcal{P}_s$. All parties in $\mathcal{P}$, bureaucrats in $\mathcal{B}$ and party $\mathcal{P}_{sender}$, send message $(\textsc{KeyGen}, sid)$ to their corresponding instance of $\mathcal{F}_{\mathsf{Sig}}$, receiving pk. Next, all parties in $\mathcal{P}$, bureaucrats in $\mathcal{B}$ and party $\mathcal{P}_{sender}$ register their signing keys by sending $(\textsc{register}, sid, \mathsf{pk})$ to $\mathcal{F}_{\mathsf{Reg}}$.

Message Input: Upon receiving a message $(\textsc{Input}, sid, ssid, \mathcal{P}_{sender}, m)$, $\mathcal{P}_{sender}$ sends $(\textsc{Encrypt}, sid, \mathcal{P}, m)$ to $\mathcal{F}_{\mathsf{CPKEPV}}$, receiving (c, π) as response. Next, $\mathcal{P}_{sender}$ sends $(\textsc{sign}, sid, c)$ to its instance of $\mathcal{F}_{\mathsf{Sig}}$, receiving $(\textsc{signature}, sid, m, \sigma_{sender})$ as response. Finally, $\mathcal{P}_{sender}$ outputs $\sigma_{sender} = (m, c, \pi, \sigma_{sender})$.

Joint Authentication: Upon receiving a message $(\textsc{Auth}, sid, ssid, \mathcal{P}_{sender}, m)$, a party $\mathcal{P}_i \in \mathcal{P}$ checks if it has received $(sid, ssid, c, \sigma_{sender})$ such that c is a ciphertext that can be correctly decrypted by $\mathcal{F}_{\mathsf{CPKEPV}}$ yielding a valid proof of plaintext knowledge and that σ_{sender} is a valid signature on c under $\mathcal{P}_{sender}$'s public-key (retrieved) from $\mathcal{F}_{\mathsf{Reg}}$ according to $\mathcal{P}_{sender}$'s instance of $\mathcal{F}_{\mathsf{Sig}}$. Formally, $\mathcal{P}_i$ proceeds as follows:

1. Send $(\textsc{Decrypt}, sid, \mathcal{P}, c)$ to $\mathcal{F}_{\mathsf{CPKEPV}}$, wait for (m, π) as response, send $(\textsc{Verify}, sid, \mathcal{P}_{owner}, c, m, \pi)$ to $\mathcal{F}_{\mathsf{CPKEPV}}$ and check that 1 is received as response.
2. Send $(\textsc{Retrieve}, sid, \mathcal{P}_{sender})$ to $\mathcal{F}_{\mathsf{Reg}}$, wait for $(\textsc{Retrieve}, sid, \mathcal{P}_{sender}, \mathsf{pk})$, send $(\textsc{verify}, sid, m, \sigma_{sender}, \mathsf{pk})$ to $\mathcal{F}_{\mathsf{Sig}}$ and check that $(\textsc{verified}, sid, m, 1)$ is received as response.

If all checks succeed, $\mathcal{P}_i$ sends $(\textsc{sign}, sid, c)$ to its instance of $\mathcal{F}_{\mathsf{Sig}}$, receiving $(\textsc{signature}, sid, m, \sigma_i)$ as response. $\mathcal{P}_i$ outputs $\sigma_i = (m, c, \pi, \sigma_i)$. Analogously, upon receiving $(\textsc{Blind-Auth}, sid, ssid, \mathcal{P}_{sender})$, bureaucrat $\mathcal{B}_j \in \mathcal{B}$ proceeds the same way as $\mathcal{P}_i$ except for not checking that c is a valid ciphertext with respect to $\mathcal{F}_{\mathsf{CPKEPV}}$ (*i.e.* skipping Step 1 of $\mathcal{P}_i$'s checks). If all checks succeed, $\mathcal{B}_j$ outputs $\hat{\sigma}_j = (c, \sigma_j)$.

[a] We abuse notation and let $\mathcal{P}_{owner}$ denote a set of parties instead of single party in $\mathcal{F}_{\mathsf{CPKEPV}}$

Fig. 15. Protocol π_{PSAuth} realizing $\mathcal{F}_{\mathsf{PSAuth}}$.

Proof. We construct a simulator $\mathcal{S}$ following the approach of the simulator for π_{CPKEPV}. Basically, $\mathcal{S}$ runs with an internal copy of the adversary $\mathcal{A}$ and forwards all communication between the environment $\mathcal{Z}$ and $\mathcal{A}$. Additionally, $\mathcal{S}$ simulates functionalities $\mathcal{F}_{\mathsf{Reg}}$, $\mathcal{F}_{\mathsf{CPKEPV}}$ and $\mathcal{F}_{\mathsf{Sig}}$ towards its internal adversary, acting exactly as in the descriptions of these functionalities, except for when explicitly mentioned. Basically, $\mathcal{S}$ simulates honest parties towards $\mathcal{A}$ by acting exactly as those honest parties would in π_{PSAuth}. When it is notified by $\mathcal{F}_{\mathsf{PSAuth}}$ that an input message query or a joint authentication query has been received, it simulates the registering and retrieval of signature keys towards $\mathcal{A}$, respectively. If $\mathcal{A}$ corrupts

Protocol π_{PSAuth} (Public Verification)

Public Verification: Upon receiving $(\textsc{Verify}, sid, ssid, \mathcal{P}_{sender}, m, \sigma_{sender}, \sigma_1, \ldots, \sigma_n, \hat{\sigma}_1, \ldots, \hat{\sigma}_b)$, a party $\mathcal{V}_i \in \mathcal{V}$ first parses all tokens $\sigma_{sender}, \sigma_1, \ldots, \sigma_n$ as (m, c, π, σ_i) and check that (m, c, π) is the same in all tokens. It then parses all tokens $\hat{\sigma}_1, \ldots, \hat{\sigma}_b$ as (c, σ_j) and checks that all c also have the same value. $\mathcal{V}_i$ then sends $(\textsc{Verify}, sid, \mathcal{P}, c, m, \pi)$ to $\mathcal{F}_{\mathsf{CPKEPV}}$ and checks that the response is 1. It then retrieves the public-keys for $\mathcal{P}_{sender}$, all parties in $\mathcal{P}$ and all bureaucrats in $\mathcal{B}$ from $\mathcal{F}_{\mathsf{Reg}}$. For all signatures σ retrieved in Step 1, $\mathcal{V}_i$ queries the $\mathcal{F}_{\mathsf{Sig}}$ instance corresponding to the party who generated the token with $(\textsc{verify}, sid, m, \sigma, \mathsf{pk})$ where pk is the public-key retrieved for that part and checks that $(\textsc{verified}, sid, m, 1)$ is received as response. If all of these checks succeed, $\mathcal{V}_i$ sets $v = 1$ (otherwise, it sets $v = 0$) and afterwards outputs $(\textsc{Verify}, sid, ssid, \mathcal{P}_{sender}, m, v)$.

Fig. 16. Protocol π_{PSAuth} realizing $\mathcal{F}_{\mathsf{PSAuth}}$ (continued).

at least one party $\mathcal{P}_i \in \mathcal{P}$ and/or $\mathcal{P}_{sender}$, acting this way allows $\mathcal{S}$ to perfectly simulate an execution of π_{PSAuth} towards corrupted bureaucrats. Notice that since $\mathcal{S}$ learns the messages that should be sent to corrupted bureaucrats from $\mathcal{A}$'s interactions with simulated $\mathcal{F}_{\mathsf{CPKEPV}}$, it can simulate π_{CPKEPV} in this case in such a way that later revealing the proofs of plaintext validity π will result in a view consistent with these messages. However, an important corner case is that when $\mathcal{A}$ corrupts all bureaucrats but not $\mathcal{P}_{sender}$ or one party $\mathcal{P}_i \in \pi$, since in this case $\mathcal{S}$ must simulate interactions between corrupted bureaucrats and $\mathcal{F}_{\mathsf{CPKEPV}}$ without knowing the committed message. In order to deal with this case, $\mathcal{S}$ deviates from the perfect simulation of $\mathcal{F}_{\mathsf{CPKEPV}}$ honest execution of protocol π_{PSAuth} and simulates interactions between corrupted bureaucrats and $\mathcal{F}_{\mathsf{CPKEPV}}$ using dummy ciphertexts (*e.g.* with random messages). Later on, after it learns the actual messages, $\mathcal{S}$ simulates the verification interface of $\mathcal{F}_{\mathsf{CPKEPV}}$ in such a way that verification queries sent to verify the dummy ciphertexts with respect to the actual messages and accompanying proofs of plaintext validity are answered positively (*i,e* simulated $\mathcal{F}_{\mathsf{CPKEPV}}$ answers with 1 when queried with $(\textsc{Verify}, sid, \mathcal{P}, c', m, \pi)$ where c' is the dummy ciphertext and (m, π) are the actual jointly authenticated message along with its proof of plaintext validity).

D Example an for Existing Protocol with an Über Simulator

An example, which we will not discuss in detail, are two-party secure computation protocols with security against malicious parties based on Garbled Circuits such as [23,13]. These protocols run between a sender $\mathcal{P}_1$ and a receiver $\mathcal{P}_2$ (where only $\mathcal{P}_2$ obtains output) as follows:

1. First, $\mathcal{P}_1$ generates multiple garbled circuits together with input keys for each such circuit. $\mathcal{P}_1$ commits to the circuit and its input keys. It then inputs the possible choices of input keys of $\mathcal{P}_2$ into Oblivious-Transfer (OT) functionalities.

2. $\mathcal{P}_2$ uses the OTs to obtain its input keys.
3. $\mathcal{P}_1$ sends the circuit and its input keys by opening the commitments.
4. $\mathcal{P}_2$ evaluates the circuits and both parties run a consistency check protocol showing that the circuits were correctly generated and the input keys are consistent.

Secure two-party computation protocols that follow the above pattern naturally have an über simulator as defined in Definition 7 as we will explain now.

The security proof usually consists of two simulators $\mathcal{S}_1$ for a corrupted sender and $\mathcal{S}_2$ for a corrupted receiver. $\mathcal{S}_1$ sends random inputs to the OT functionalities and otherwise only checks that the garbling information was generated correctly by the malicious $\mathcal{P}_1$ (and extracts the inputs of $\mathcal{P}_1$). Usually, $\mathcal{S}_1$ uses a random input to the OTs and otherwise just follows the protocol. As the OT functionalities by their security guarantee hide the input of $\mathcal{P}_2$, a $\mathcal{S}_1$ instance using the actual inputs of $\mathcal{P}_2$ will have Simulation and Execution consistency of the transcripts.

In case of the simulator $\mathcal{S}_2$ the standard strategy is to first extract the input x_2 of the malicious $\mathcal{P}_2$ using the OT functionalities, then to obtain the output y from the ideal functionality $\mathcal{F}$, to choose a random input $\tilde{x}_1$ and finally to simulate the garbled circuits in such a way that they outpus y for input keys derived from $\tilde{x}_1, x_2$. This clearly is not consistent as required by the definition, but we can instead define a simulator $\mathcal{S}_2'$ which would just run the actual protocol based on the input x_1 that it now has. By the security argument of the original protocol, this is both Simulation consistent and by running the actual protocol, also Execution consistent.