Public Verifiability For Composable Protocols Without Adaptivity Or Zero-Knowledge

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

[1] Aarhus University
cbaum@cs.au.dk
[2] IT University Copenhagen
bernardo@bmdavid.com
[3] Monash University rafael@dowsley.net

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 the 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 cannot directly be proven secure. In this work we study Non-Interactive Public Verifiability of UC protocols in a practical setting, i.e. without requiring adaptively secure primitives or heavy computational tools such as NIZKs. As Non-Interactive Public Verifiability is crucial when composing protocols with a public ledger, our approach can be beneficial when designing these with formal security and practicality in mind. We discuss and formalize what Non-interactive Public Verifiability means in the Universal Composability Framework and construct an efficient transformation that achieves this notion for a large class of practical cryptographic protocols.

1 Introduction

Universal Composability (UC) [13] is currently the most popular framework for designing and proving security of cryptographic protocols under arbitrary composition. It allows one to prove that a protocol remains secure even in complex scenarios consisting of multiple nested protocol executions. 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

* Work partially done while Carsten Baum and Rafael Dowsley were at the Bar Ilan University and 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 partially supported by a grant from Concordium Foundation and by DFF grants number 9040-00399B (TrA^2C) and and number 9131-00075B (PUMA).

security is argued on a very high level only. This is partially due to the complexity of fully expressing a protocol in UC, but also because achieving UC security appears to be overly complicated for some, seemingly simple and intuitive cases.

One such case is that of public verifiability, a property that allows third party verifiers to check that a protocol execution was successful and yielded a certain output (or that it aborted). This property is particularly important in the setting of decentralized systems and public ledgers (*i.e.* blockchains [33,9,27,29,26,22]), where new parties can join an ongoing protocol execution on-the-fly after verifying that their view of the protocol is valid. Public verifiability also plays a central role in a recent line of research [2,8,30,5] on secure multiparty computation [4] (MPC) protocols that rely on a public ledger to achieve fairness [5] by penalizing cheating parties, circumventing fundamental impossibility results [23]. Furthermore, it is an intrinsic property of randomness beacons [20,21], a central component of provably secure Proof-of-Stake blockchain protocols [30,5,22]. However, most of these works achieve public verifiability by relying on heavy tools such as non-interactive zero knowledge proof systems and strong assumptions such as adaptive security of the underlying protocols (with the notable exception of [5], which avoids these by painstakingly redefining and reproving security of publicly verifiable versions of known protocols).

1.1 The Problems of Achieving Verifiability in UC

In order to motivate our work, we now 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 $\mathcal{F}$, so that it also has 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 were provided by a party $\mathcal{P}_i$ to $\mathcal{F}^{\mathtt{V}}$ and that certain outputs were sent from $\mathcal{F}^{\mathtt{V}}$ to $\mathcal{P}_i$. To achieve this, we design a protocol $\Pi^{\mathtt{V}}$ based on the original protocol Π that implements these interfaces.

The first step in any such solution is to construct $\Pi^{\mathtt{V}}$ in such a way that each party commits to inputs and randomness that it will use in the protocol Π. The parties then run Π based on these committed values and exchange messages in an authenticated way. Let us assume that we are okay with revealing the inputs after Π is completed (we will discuss this assumption later). Intuitively, this should yield a simple verification procedure: each involved party can inspect the committed inputs and randomness of all other parties, re-run these parties in its head and compare its simulated messages to the signed protocol transcript. A third party could then do the same, based on the commitments and a transcript of Π. Unfortunately, even just using this simple approach means that we run

[4] Protocols that allow mutually distrustful parties to compute joint functions on private inputs without revealing them.

[5] Ensuring either all parties obtain the protocol output or nobody does, including the adversary.

into adaptivity problems when trying to prove $\Pi^{\mathtt{V}}$ secure. Similar problem have been observed in e.g. [28] and we explain it more in detail using an example.

Consider a two-party secure computation protocol (2PC) with active security based on Garbled Circuits (GCs) such as [32,17]. 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 circuits and their input keys. It inputs the available input keys of $\mathcal{P}_2$ into Oblivious-Transfer (OT) functionalities.
2. $\mathcal{P}_2$ uses the OTs to obtain its input keys.
3. $\mathcal{P}_1$ decommits the circuits and its input keys towards $\mathcal{P}_2$.
4. $\mathcal{P}_2$ evaluates the circuits and both parties run a consistency check showing that most circuits were correctly generated and the input keys are consistent.

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 $\mathcal{P}_1$). 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 functionality, then to obtain the output y from the ideal functionality $\mathcal{F}$ of the protocol, to choose a random input $\tilde{x}_1$ and finally to simulate the garbled circuits in such a way that they output y for input keys derived from $\tilde{x}_1, x_2$.

In order to have a verifiable version of the aforementioned protocol, observe that $\mathcal{F}^{\mathtt{V}}$ will release the actual input x_1 of $\mathcal{P}_1$ after the computation has finished. But in $\mathcal{S}_2$ we generated the circuit in such a way that for an assumed dummy input $\tilde{x}_1$ it must output y, and the garbling itself may not even be a correct garbling of the function in question. That means that there might not exist randomness to explain the output of $\mathcal{S}_2$ in a security proof of the verification. More generally, for verification of an instance of Π 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 $\mathcal{F}^{\mathtt{V}}$ given the randomness of the party. That implies that Π has to be adaptively secure to begin with if we want to prove that $\Pi^{\mathtt{V}}$ UC-securely implements $\mathcal{F}^{\mathtt{V}}$ and that the implementations of $\mathcal{F}^{\mathtt{V}}$ in practice will be slow, since the number of protocols Π which are adaptive is limited. This seems counter-intuitive: beyond the technical reason to allow UC simulation, we see no explanation why only adaptive protocols should be verifiable in the aforementioned way. It appears counter-intuitive why the aforementioned static 2PC protocol could not be proven to have such verifiability[6].

1.2 Our Contributions

In this work, we show that we can compile a large class of statically secure UC-secure protocols realizing a given UC functionality into protocols realizing a publicly verifiable version of the functionality, which allows a party to non-interactively prove that it obtained a certain output upon revealing its input.

[6] Obviously one can achieve the result, even without revealing inputs, by using expensive generic NIZKs [28,30]. Our goal is to avoid these.

On one hand, we have the caveat of requiring the party who generates the proof to open its input. On the other hand, this allows us to circumvent the need for expensive generic zero knowledge proofs and adaptive security (as needed in the compilers of [28,30]), introducing a technique that can compile protocols relying only on cheap commitments and "joint authentication" functionalities which can be realized using cheap public-key primitives. Moreover, the need to reveal the prover's real input can also be solved in some cases by executing protocols with dummy parties (with dummy inputs). This e.g. works for MPC where each real party would additionally simulate another party that would also join the computation. This party would be without input but would obtain the output. It can then later serve as a prover, avoiding sacrificing the input of the real party at the cost of higher complexity.

Our approach in its current form is with protocols that UC-realize functionalities that have 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. We 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 and might be of independent interest. It would be interesting future work to extend our techniques to reactive computations.

Our results can be concretely applied in the setting of decentralized systems (*e.g.* blockchains [33,9,27,29,26,22]) and protocols based on blockchain-style public ledgers [2,8,30,5,20,21], serving both as foundational public verifiability definitions and as a tool for neatly achieving this property. For example, using our techniques, it would be possible to construct the publicly verifiable building blocks of Insured MPC [5] without individually redefining each functionality and reproving each protocol's security. In particular, our results have already been used as an essential tool in follow-up work constructing UC randomness beacons [21].

1.3 Our Approach

Our route to black-box verifiability. We construct a compilation method to generically achieve public verifiability for protocols that follow the outline described in Section 1.1 and formalized in Section 2. For this, we start with an observation similar to [28], namely that by fixing the inputs, randomness and messages in a protocol we can get guarantees about the outputs. This is because fixing the inputs, randomness and received messages essentially fixes the view of a party, as the messages generated and sent by a party are deterministic given all of these other values. Therefore, our main idea is to fix parties' input and randomness pairs by having parties commit to these pairs and authenticate the messages exchanged between parties in such a way that an external party can verify such committed/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 and might be overkill for some protocols. We formalize this concept to the notion of *transcript non-malleability* that is defined in Section 3.1. The hope is that we might not have all exchanged messages fixed for some protocols, but an adversary that e.g. is allowed to replace all those messages exchanged between dishonest parties after the fact, or those that were exchanged with the honest parties, does not have enough leverage to forge a fully consistent transcript for a different output.

Proving security in UC. 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 to be a too restrictive requirement. In Section 3.2 we address this problem using *input-aware simulators* (or über simulators), which can be parameterized with specific inputs for the simulated honest parties, generating simulated transcripts consistently with these inputs. By embedding these über simulators into publicly verifiable functionalities as described in Section 3.4, we are able to essentially delegate the simulation of the original protocol being compiled to the functionality's internal über simulator while focusing on the public verifiability machinery in our simulators for the compiled protocol/functionality. As these über simulators form the core technical contribution of our work, we will now motivate their definition and use in more detail.

A first idea for proving security of the compiled protocol $\Pi^{\mathtt{V}}$ is to let the simulation of Π be done inside $\mathcal{F}^{\mathtt{V}}$, the compiled functionality that is verifiable, using something similar to the simulator of Π that simulates an execution with the adversary using the actual honest party inputs for the simulated honest parties. We would give $\mathcal{F}^{\mathtt{V}}$ an extra "next message function"-style interface $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$ accessible to the ideal adversary. The simulator of the compiled protocol $\Pi^{\mathtt{V}}$ will be able to use $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$ in order to generate the protocol messages in the simulation, allowing us to easily explain Π consistently after the fact. $\mathcal{F}^{\mathtt{V}}$ can provide $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$ 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 with the adversary and functionalities in the actual proof as well as simulation of the compilation machinery. Based on the UC security of Π, making $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$ available will not break existing constructions. This is because before we open any inputs or randomness, it is indistinguishable if we simulate an execution with the actual inputs of honest parties running Π or random inputs due to the UC-security of Π. The "external" simulator for $\Pi^{\mathtt{V}}$ in the security proof then extracts the protocol messages of Π that the adversary sends (and those sent to the hybrid functionalities) and inputs these accordingly into $\mathcal{F}^{\mathtt{V}}$'s $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$ interface.

Considering our example, $\mathcal{S}_1$ 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 $\mathcal{S}_1$ simulation inside $\mathcal{F}^{\mathtt{V}}$ using the actual inputs of $\mathcal{P}_2$ would still be indistinguishable. For the other party, we can construct 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, the distribution of this should be indistinguishable from the output of $\mathcal{S}_2$.

But just simulating based on real inputs of the honest parties is not sufficient: 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} does not know the semantics of the original security proof of Π and can therefore not extract the inputs from $\mathcal{A}$. In the example of 2PC, the outer simulator would have to know that it needs to extract the inputs of the dishonest party from the OT functionality, but such knowledge is specific for this class of protocols. Instead, we would like the outer simulator to simply extract inputs committed by its copy of the adversary and check that it matches the transcript in the adversary's interaction with the internal simulator (checking that verification succeeds or not).

To resolve this problem, we require that for our compiled protocols Π there exists a special "über" simulator (defined fully in Section 3.2) which addresses these problems of simulation and extraction accordingly. As can be seen from our example, an efficient über simulator must not be artificial and strong, but could possibly simply be constructed. Its requirement also differs from requiring adaptivity of the protocol, as we at no point changed any security requirement about the 2PC protocol. And the strategy is not just bound to GC-based protocols, as e.g. many MPC protocols such as [34,25] simulate their online phase anyway in the security proof using "artificial" fixed inputs and otherwise run the protocol honestly while they are still able to extract. We can therefore directly make those protocols verifiable, as we will indeed show in Section 3.3.

How to realize transcript non-malleability. In order to construct compilers from Π to Π^{V} we have to use cryptographic tools that help us achieve a certain degree of transcript non-malleability. For this, we require parties to use what we call "joint authentication". Joint Authentication works for both public and private messages. In the public case, performing joint authentication is achieved by simply having all parties sign a message sent by one of them. In the private case, we essentially allow parties to authenticate commitments to private messages that are only opened to certain parties. Later on, any of the parties who did receive that private message (*i.e.* the opening of the commitment to the message) can publicly prove that they indeed obtained a certain message that had been jointly authenticated by all the other parties involved in the protocol execution. More importantly, joint authentication does not perform any communication on itself but provides authentication tokens that can be verified in a non-interactive manner. These public and secret joint authentication functionalities allow us to force dishonest parties to commit to their transcript without revealing private messages ahead of time or implying consensus on its own (since parties are still required to send their messages and authentication tokens through regular channels). Joint authentication, by requiring parties to acknowledge receipt, forces the compiled protocol into synchronous form, which is why we only consider synchronous protocols to begin with. We define joint authentication functionalities in Section 4 and show how to realize them in Supplementary Material C.

For our example protocol, this would mean that both $\mathcal{P}_1, \mathcal{P}_2$ initially commit to their inputs and randomness. Additionally, they will both sign each message that they send and those messages that they receive and that have been signed by the sender. This will make it possible for any third party to later verify correctness of these values.

Putting things together. We use the techniques described above to compile a protocol Π that fits one of the levels of our transcript non-malleability definition and realizes a functionality $\mathcal{F}$ in the $\mathcal{F}_1, \ldots, \mathcal{F}_n$-hybrid model into a protocol $\Pi^{\mathtt{V}}$ that realizes a publicly verifiable version of $\mathcal{F}$ called $\mathcal{F}^{\mathtt{V}}$ in the $\mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_n^{\mathtt{V}}$-hybrid model (*i.e.* assuming that the setup functionalities are also publicly verifiable). Our compilation technique has two main components: 1. use a special instance of secret joint authentication to commit to and authenticate each party's input and randomness pairs as defined in Π; 2. execute Π and use public/secret joint authentication to jointly authenticate each message of Π exchanged among them. The first step fixes the input and randomness pairs of each party in such a way that the parties can publicly prove they used a certain input and randomness pair at a later point with a non-interactive proof. The second step makes sure that the whole protocol transcript of executing Π is fixed with a similar guarantee that parties can later publicly prove that a given transcript was obtained (after revealing the private messages they receive) with a non-interactive proof. Notice that using these guarantees we have basically brought Π to a very strong level of transcript non-malleability, since the adversary cannot lie about its input and randomness pairs nor its view of the transcript. In order to realize the public verifiability interface of $\mathcal{F}^{\mathtt{V}}$, we have a party open its input and randomness pair as well as its view of the transcript, which could not have been forged, allowing the verifier to execute an honest party's steps in Π in order to verify that a given output is obtained.

As discussed before, when proving security of this compiler, we have to overcome the hurdle of simulating verification when an honest party opens its input and randomness pair. We do that by delegating the simulation of the original steps of Π to an über simulator for Π embedded in $\mathcal{F}^{\mathtt{V}}$. Essentially, our simulator $\mathcal{S}$ will simulate an execution with an internal copy of the adversary by extracting all Π messages sent by this adversary through the simulated joint authentication functionalities and forwarding these messages to the über simulator $\mathcal{S}^{\mathtt{U}}$ for Π embedded into $\mathcal{F}^{\mathtt{V}}$ through interface $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$ (and forwarding back $\mathcal{S}^{\mathtt{U}}$'s answers to the adversary). If an honest party activates public verification and reveals its input, we are now guaranteed that the transcript of $\mathcal{S}$'s simulated execution is consistent with that honest party's input, since it was generated by $\mathcal{S}^{\mathtt{U}}$ embedded into $\mathcal{F}^{\mathtt{V}}$ and parameterized with honest party inputs.

To compile our example protocol, we now combine all of the aforementioned steps and additionally assume that the OT-functionality as well as the commitment-functionality are verifiable on their own. By the compiler theorem, this will mean that the resulting protocol is verifiable according to our definition.

1.4 Related Work

Despite being very general, UC has seen many extensions such as e.g. UC with joint state [19] or Global UC [15], aiming at capturing protocols that use global ideal setups. Verifiability for several kinds of protocols has been approached from different perspectives, such as cheater identification [28,6], verifiability of MPC [4,36], incoercible secure computation [1], secure computation on public ledgers [2,8,30,31], and improved definitions for widely used primitives [12,11]. Another solution to solve the adaptivity requirement was recently presented in [7], but their approach only works for functionalities without input. A different notion of verifiability was put forward in publicly verificable covert 2PC protocols such as [3] and its follow-up works, where parties can show that the other party has cheated. However, most of these works require adaptive security of the underlying protocol or zero knowledge proof systems. A notable exception is the publicly verifiable MPC protocol of [5], which avoids these issues by painstakingly redefining each functionality used as a building block and reproving security of publicly verifiable versions of known protocols realizing these functionalities. To the best of our knowledge, no previous work has considered a generic definition of non-interactive public verifiability in the UC framework nor a black-box compiler for achieving such a notion *without* requiring adaptive security of the underlying protocol or zero knowledge proof systems.

2 Preliminaries

We denote the security parameter by κ and the concatenation of two strings a and b by $a \,\|\, b$. Let $y \xleftarrow{\$} F(x)$ 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 $y \leftarrow F(x; r)$. $y \leftarrow F(x)$ is used for a deterministic algorithm. For a set $\mathcal{X}$, let $x \xleftarrow{\$} \mathcal{X}$ 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 denote by $\mathsf{negl}(\kappa)$ the set of negligible functions of κ and 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 $|\Pr[\mathcal{D}(X_{\kappa,z}) = 1] - \Pr[\mathcal{D}(Y_{\kappa,z}) = 1]|$ 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*, denoted by $X \approx_c Y$.

2.1 Secure Protocols

A protocol Π consists of the algorithms $\texttt{nmes}, \texttt{out}$ 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:

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

`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 $\bot$. The values x_i, r_i might not be necessary in every protocol and we allow use of `out` without it as well.

`nmes` generates two different types of messages, namely `m`- and `mres`-messages. As we shall see later, the `m`-messages are used for communication *among parties* whereas `mres`-messages are exchanged *between a party and a functionality*. Therefore, each `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 `nmes` may either be an actual string or a symbol $\bot$, 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.

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 κ). If these will be sent through point-to-point secure channels, then 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. We do not restrict the output $\mathtt{y}_i^{(\tau)}$ which each party obtains in the end of the computation and which should be verifiable. This permits the general setting where all the $\mathtt{y}_i^{(\tau)}$ might be completely different. This is the standard for many interesting functions that one can compute, e.g. Oblivious Transfer.

2.2 Universal Composition of Secure Protocols

In this work we use the (Global) Universal Composability or (G)UC model [13,15] for analyzing security and refer interested readers to the original works for more

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 $\mathcal{P}_j$. For each $q \in [r]$ send $\mathtt{mres}_{i,q}^{(\tau,\rho)}$ 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 $\mathcal{P}_i$ computes and outputs $\mathtt{y}_i^{(\tau)} \leftarrow \mathtt{out}(i, x_i, r_i, s, \tau, \mathcal{M}_{\cdot,i}, \mathcal{N}_{\cdot,i})$.

Fig. 1. The generic protocol Π.

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).

Protocols 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 $I \subset \mathcal{P}$. Parties can exchange messages with each other and also with resources, which we call *ideal functionalities* (which themselves are PPT iTMs). To simplify notation we assume that the messages between parties are sent over secure channels.

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 **Input$_i$**. While **Input$_i$**, **Output$_i^{(\tau)}$** are fixed in their semantics, the application may freely vary how **Compute$^{(\tau)}$** may act upon the inputs or generate out-

Functionality $\mathcal{F}$

Functionality $\mathcal{F}$ has common public input s and interacts with a set $\mathcal{P}$ of n parties and an ideal adversary $\mathcal{S}$. Upon initialization, $\mathcal{S}$ is allowed to corrupt a set $I \subset \mathcal{P}$ of parties where $|I| \leq k$ and $k < n$. Each of $\mathcal{F}$'s interfaces falls into one of 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 $J_\tau \subseteq \mathcal{P}$ 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 these.
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 $\mathcal{F}$.

puts. The only constraint that we make is that for each of the $\tau \in [G]$ rounds, **Compute**$^{(\tau)}$ sets output values $(\mathtt{y}_1^{(\tau)}, \dots, \mathtt{y}_n^{(\tau)})$.

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.

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 [14], the key registration functionality $\mathcal{F}_{\mathsf{Reg}}$ from [16] and an authenticated bulletin board functionality $\mathcal{F}_{\mathsf{BB}}$, which are described in Supplementary Material 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 Supplementary Material B.

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 $\mathcal{F}$. $\mathcal{F}^{\mathtt{V}}$ is parameterized by an über simulator $\mathcal{S}^{\mathtt{U}}$ executed internally (as discussed in Section 3.4) and maintains binary variables $\textsf{verification-active}, \textsf{verify-1}, \ldots, \textsf{verify-n}$ that are initially 0 and used to keep track of the verifiable outputs. Apart from the set of parties $\mathcal{P}$ and ideal adversary $\mathcal{S}$ defined in $\mathcal{F}$, $\mathcal{F}^{\mathtt{V}}$ interacts with verifiers $\mathcal{V}_i \in \mathcal{V}$.

Register Verifier (private): Upon receiving $(\textsc{Register}, sid)$ from $\mathcal{V}_i$:
- If $\textsf{verification-active} = 1$ send $(\textsc{Register}, sid, \mathcal{V}_i)$ to $\mathcal{S}$. If $\mathcal{S}$ answers with $(\textsc{Register}, sid, \mathcal{V}_i, ok)$, set $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_i$ and return $(\textsc{Registered}, sid)$ to $\mathcal{V}_i$.
- If $\textsf{verification-active} = 0$ return $(\textsc{Verification-Inactive}, sid)$ to $\mathcal{V}_i$.

Register Verifier (public): Upon receiving $(\textsc{Register}, sid)$ from $\mathcal{V}_i$:
- If $\textsf{verification-active} = 1$ set $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_i$ and return $(\textsc{Registered}, sid)$ to $\mathcal{V}_i$.
- If $\textsf{verification-active} = 0$ return $(\textsc{Verification-Inactive}, sid)$ to $\mathcal{V}_i$.

Activate Verification: Upon receiving $(\textsc{Activate-Verification}, sid, \textsf{open-i}, \textsf{open-input-i})$ from each $\mathcal{P}_i$ and if $\textbf{Compute}^{(1)}, \ldots, \textbf{Compute}^{(G)}$ succeeded:
1. Let $Y \leftarrow \{j \in [n] \mid \textsf{open-j} = 1 \wedge \textsf{verify-j} = 0\}$. If $Y = \emptyset$ then return.
2. Set $\textsf{verification-active} \leftarrow 1$ (if it is not set already) and deactivate the interfaces $\textbf{Compute}^{(\tau)}$ for all $\tau \in [G]$.
3. If $\textsf{open-input-i} = 1$, then set $z_i = x_i$; otherwise $z_i = \perp$.
4. Send $(\textsc{Activating-Verification}, sid, Y, \{z_j, \mathtt{y}_j^{(\tau)}\}_{j \in Y, \tau \in [G]})$ to $\mathcal{S}$. If $\mathcal{P}_i$ is honest, append its randomness R_i (obtained from $\mathcal{S}^{\mathtt{U}}$) to this message.
5. Upon receiving $(\textsc{Activating-Verification}, sid, ok)$ from $\mathcal{S}$ set $\textsf{verify-j} \leftarrow 1$ for each $j \in Y$. Then return $(\textsc{Verification-Activated}, sid, Y, \{z_j, \mathtt{y}_j^{(\tau)}\}_{j \in Y, \tau \in [G]})$ to all parties in $\mathcal{P}$.

Verify$_j$: Upon receiving $(\textsc{Verify}, sid, j, a, b^{(1)}, \ldots, b^{(G)})$ from $\mathcal{V}_i$ where $\mathcal{V}_i \in \mathcal{V}$ and $\mathcal{P}_j \in \mathcal{P}$ do the following:
- if $\textsf{verify-j} = 1$ 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 $(\textsc{Verify}, sid, j, f, B)$ to $\mathcal{V}_i$.
- If $\textsf{verify-j} = 0$ then send $(\textsc{Cannot-Verify}, sid, j)$ to $\mathcal{V}_i$.

Input$_i$: On input $(\textsc{Input}, sid, x_i)$ by $\mathcal{P}_i$ and $(\textsc{Input}, sid)$ by all other parties, forward $(\textsc{Input}, sid, x_i)$ to $\mathcal{F}$ and also forward responses from $\mathcal{F}$ to $\mathcal{P}_i$. Finally, after receiving $(\textsc{Input}, sid, x_i)$ from all $\mathcal{P}_i, i \in \overline{I}$ (*i.e.* all honest parties), initialize $\mathcal{S}^{\mathtt{U}}$ parameterizing it with x_i for all honest $\mathcal{P}_i$.

NMF$_{\mathcal{S}^{\mathtt{U}}}$: Upon input $(\textsc{NextMsgP}, sid, j, \tau, \rho, \{\mathtt{m}_{i,j}\}_{i \in I})$ where $j \in \overline{I}$ or $(\textsc{NextMsgF}, sid, q, \tau, \rho, \mathtt{mres}_{i,q})$ where $i \in I$ and $q \in [r]$ by $\mathcal{S}$, send the respective message to $\mathcal{S}^{\mathtt{U}}$. Forward all messages between $\mathcal{S}^{\mathtt{U}}$ and $\mathcal{F}$, so that $\mathcal{S}^{\mathtt{U}}$ mediates interaction between $\mathcal{F}$ and $\mathcal{S}$, also delivering extracted adversarial inputs. Finally, after $\mathcal{S}^{\mathtt{U}}$ outputs a response $(\textsc{NextMsgP}, sid, j, \tau, \rho + 1, \{\mathtt{m}_{j,i}\}_{i \in I})$ or $(\textsc{NextMsgF}, sid, q, \tau, \rho + 1, \mathtt{mres}_{q,i})$, forward it to $\mathcal{S}$.

Fig. 3. The Functionality wrapper $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. The modifications to interface **Input**$_i$ and the new interface **NMF**$_{\mathcal{S}^{\mathtt{U}}}$ are discussed in Section 3.4.

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 $\mathcal{F}^{\mathbb{V}}$ 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 $\mathbf{Output}_i^{(\tau)}$ (that we 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 $\mathbf{Verify}_i$. The modifications to $\mathbf{Input}_i$ and the new interface $\mathbf{NMF}_{\mathcal{S}^{\mathfrak{v}}}$ are related to the über simulators discussed in Section 3.4.

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, $\mathcal{F}^{\mathbb{V}}$ might implement only **Register Verifier (public)** or only **Register Verifier (private)**, respectively. We do not require $\mathcal{F}^{\mathbb{V}}$ to implement both of these interfaces, and thus define the properties of $\mathcal{F}^{\mathbb{V}}$ 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* $\mathbf{Verify}_j$ *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 Public Verifiability

We now present our approach for making protocols non-interactively verifiable. For this, we will first introduce a classification for the robustness of a protocol to attacks on its "inherent" verifiability. Then, we describe properties that are necessary to achieve simulation-based security for our approach to verifiability.

3.1 Transcript Malleability of Protocols

Informally, our approach to verification (as outlined in Section 1.3) is to leverage properties for verifiability that are potentially already built into the protocol. This is because we only want to rely on the protocol itself in a black-box fashion. As the verifier can then only rely on the protocol transcript, let us consider how such a transcript comes into existence.

In practice, we would first run a protocol instance of Π with an adversary $\mathcal{A}$. Afterwards, the adversary may have the possibility to change parts of the protocol transcript in order to trigger faulty behavior in the outputs of parties. If the adversary cannot trigger erroneous behavior, then this means that we can establish correctness of an output of such a protocol by using the messages of its transcript, some opened inputs and randomness as well as some additional properties of $\Pi = (\mathtt{nmes}, \mathtt{out})$.

If our verification therefore relies on the transcript of a protocol, then a first sign of incorrectness is if messages that a party $\mathcal{P}_i$ claims to have sent were not received by another party $\mathcal{P}_j$, if messages to and from a NIV functionality $\mathcal{F}^{\mathtt{V}}$ were not actually sent or received by $\mathcal{P}_i$ or if, in case a party $\mathcal{P}_i$ reveals both its inputs x_i and randomness r_i, the messages $\mathcal{P}_i$ claims to have sent are inconsistent with x_i, r_i when considering $\mathtt{nmes}$ and previously obtained messages.

Towards formalizing this, we denote the set of input-revealing parties as RIR. For $\mathcal{M}_{i,\cdot}, \mathcal{M}_{\cdot,i}, \mathcal{N}_{i,\cdot}, \mathcal{N}_{\cdot,i}$ we use the same syntax as in Section 2.1.

Definition 4 (Transcript Validity). *Let n be the number of parties and $\mathsf{RIR} \subseteq [n]$. For $i \in \mathsf{RIR}$ let x_i be inputs and r_i be a randomness string. Let furthermore s be an auxiliary input, $\mathcal{F}_1^{\mathtt{V}}, \ldots, \mathcal{F}_r^{\mathtt{V}}$ be a set of NIV resources and $\mathcal{M}_{\cdot,i}, \mathcal{M}_{i,\cdot}, \mathcal{N}_{\cdot,i}, \mathcal{N}_{i,\cdot}$ be those sets of messages that were defined before.*

We say that the transcript of Π is valid *if and only if*

1. *For each $i, j \in [n]$ the sets $\mathcal{M}_{i,\cdot}, \mathcal{M}_{\cdot,j}$ are consistent, meaning that each message in $\mathcal{M}_{i,\cdot}$ sent by $\mathcal{P}_i$ was received by $\mathcal{P}_j$ in $\mathcal{M}_{\cdot,j}$ and vice versa.*
2. *For each $q \in [r], i \in [n]$ $\mathcal{N}_{\cdot,i}$ are consistent with the messages that $\mathcal{P}_i$ should have obtained from $\mathcal{F}_q^{\mathtt{V}}$ via the verification interface. If $\mathcal{F}_q^{\mathtt{V}}$ allows the verification of inputs from $\mathcal{P}_i$, $\mathcal{N}_{i,\cdot}$ is consistent with $\mathcal{F}_q^{\mathtt{V}}$ as well.*
3. *For each $i \in \mathsf{RIR}, \tau \in [G]$ and $\rho \in [H_\tau]$ the sets $\mathcal{M}_{i,\cdot}, \mathcal{N}_{i,\cdot}$ are consistent with the output of $\mathtt{nmes}(i, x_i, r_i, s, \tau, \rho, \mathcal{M}_{\cdot,i}, \mathcal{N}_{\cdot,i})$.*

In a formal sense, tampering of an adversary with the transcript would be ok unless it leads to two self-consistent protocol transcripts with outputs $\widehat{\mathtt{y}}_i^{(\tau)} \neq \mathtt{y}_i^{(\tau)}$ for some $\mathcal{P}_i$ such that both $\widehat{\mathtt{y}}_i^{(\tau)}, \mathtt{y}_i^{(\tau)} \neq \perp$. To achieve this, transcript validity is a necessary, but not a sufficient condition. For example, if no messages or inputs or randomness of any party are fixed, then $\mathcal{A}$ could easily generate two correctly distributed transcripts for different outputs that fulfill this definition using the standard UC simulator of Π.

We now define the security game that allows us to further constrain $\mathcal{A}$ beyond transcript validity. In it, we rely on fixing certain parts while the transcript is generated: an adversary $\mathcal{A}$ will first run the protocol with a challenger $\mathcal{C}$

that simulates honest parties whose inputs and randomness $\mathcal{A}$ does not know (initially). Upon completion of this protocol, the adversary will first obtain some additional potentially secret information of the honest parties, upon which it outputs two valid protocol transcripts. $\mathcal{A}$ will win if the transcripts coincide in some parts with the interactive protocol that $\mathcal{A}$ ran with $\mathcal{C}$, 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. We consider scenarios regarding: (1) whether the dishonest parties can change their inputs and randomness after the execution (parameter ν); (2) what is the set of parties RIR 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).

The parameters ν, RIR have the following impact: if $\nu = \mathsf{ncir}$ then the dishonest parties *are not committed* to the input and randomness in the beginning of the execution. Anything that is revealed from parties in $I \cap \mathsf{RIR}$ might be altered by the adversary. If instead $\nu = \mathsf{cir}$ then all parties *are committed* to the input and randomness in the beginning of the execution. That means that the adversary cannot alter inputs or randomness of honest or dishonest parties from RIR, i.e. of those parties whose x_i, r_i are revealed for verification.

For μ we give the adversary the following choices:

$\mu = \mathsf{ncmes}$**:** $\mathcal{A}$ 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 formalize transcript non-malleability as follows:

Definition 5. *Let Π be a synchronous protocol that is secure against a static adversary corrupting up to $k < n$ parties using r NIV resources. For $\nu \in \{\mathsf{cir}, \mathsf{ncir}\}$, $\mu \in \{\mathsf{ncmes}, \mathsf{chsmes}, \mathsf{chmes}, \mathsf{cmes}\}$ and $\mathsf{RIR} \subseteq [n]$, we define the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$:*

1. *Both $\mathcal{A}, \mathcal{C}$ obtain s. $\mathcal{C}$ sets up instances $\mathcal{F}_1^{\mathsf{V}}, \ldots, \mathcal{F}_r^{\mathsf{V}}$.*
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 $\{x_i, r_i\}_{i \in \overline{I}}$. If $\nu = \mathsf{cir}$, $\mathcal{A}$ sends $\{x_j, r_j\}_{j \in I}$ to $\mathcal{C}$.*
4. *$\mathcal{C}$ runs an instance of the protocol Π with $\mathcal{A}$. In each round of Π $\mathcal{C}$ first computes the messages of all honest parties $\mathcal{P}_i \in \overline{I}$ using $\mathtt{nmes}$ and sends these to both $\mathcal{A}$ as well as $\mathcal{F}_q^{\mathsf{V}}$. Then $\mathcal{A}$ interacts with all instances of $\mathcal{F}_q^{\mathsf{V}}$ and sends messages of dishonest parties addressed to the honest parties to $\mathcal{C}$. If $\mu = \mathsf{cmes}$ then $\mathcal{A}$ must also send messages exchanged between dishonest parties. Finally, $\mathcal{C}$ stores all those messages sent to as well as messages received from $\mathcal{A}$ in $\mathcal{M}_{\cdot,i}, \mathcal{M}_{i,\cdot}$ (if $\mu = \mathsf{cmes}$ also those sent between dishonest parties). It stores all messages that an honest $\mathcal{P}_i \in \overline{I}$ received from $\mathcal{F}_q^{\mathsf{V}}$ in $\mathcal{N}_{\cdot,i}$ and those that it sent to input-verifiable $\mathcal{F}_q^{\mathsf{V}}$ in $\mathcal{N}_{i,\cdot}$.*
5. *$\mathcal{C}$ sends $\{x_i, r_i, \mathcal{N}_{\cdot,i}\}_{i \in \overline{I} \cap \mathsf{RIR}}$ to $\mathcal{A}$. For $\mathcal{P}_i \in \overline{I} \setminus \mathsf{RIR}$ it sends $\mathcal{M}_{i,\cdot}, \mathcal{M}_{\cdot,i}, \mathcal{N}_{i,\cdot}, \mathcal{N}_{\cdot,i}$.*

6. $\mathcal{A}$ *sends two protocol transcripts* Π^0, Π^1 *including inputs* x_i^b, r_i^b *for* $i \in \mathsf{RIR}$ *and messages* $\mathcal{M}_{i,\cdot}^b, \mathcal{M}_{\cdot,i}^b$ for all parties $i \in [n]$ *and* $b \in \{0,1\}$. $\mathcal{C}$ *checks that*
   (a) *Both transcripts* Π^0, Π^1 *are consistent according to Definition 4.*
   (b) *If* $\nu = \mathsf{cir}$ *then* $r_i^b = r_i$ *and* $x_i^b = x_i$ *for* $i \in \mathsf{RIR}$. *If instead* $\nu = \mathsf{ncir}$ *then* $r_i^b = r_i$ *and* $x_i^b = x_i$ *for* $i \in \overline{I} \cap \mathsf{RIR}$.
   (c) $\mathcal{N}_{\cdot,i}^b$ *for* $i \in [n]$ *is consistent with* $\mathcal{F}_q^{\mathsf{V}}$. *Moreover, for each* $\mathcal{P}_i$ *where* $\mathcal{F}_q^{\mathsf{V}}$ *reveals inputs of* $\mathcal{P}_i$ *check if* $\mathcal{N}_{i,\cdot}^b$ *is consistent with* $\mathcal{F}_q^{\mathsf{V}}$.
   (d) *If* $\mu = \mathsf{cmes}$, $\mathcal{M}_{i,j}^b = \mathcal{M}_{i,j}$ *for all* $i, j \in [n]$.
   (e) *If* $\mu = \mathsf{chmes}$, $\mathcal{M}_{i,j}^b = \mathcal{M}_{i,j}$ *for all* $i, j \in [n]$ *where either* $i \in \overline{I}$ *or* $j \in \overline{I}$.
   (f) *If* $\mu = \mathsf{chsmes}$, $\mathcal{M}_{i,j}^b = \mathcal{M}_{i,j}$ *for all* $j \in [n], i \in \overline{I}$.

   If not, then $\mathcal{C}$ *outputs* 0.
7. $\mathcal{C}$ *outputs* 1 *if either there exists* $i \in \mathsf{RIR}, \tau \in [G]$ *such that*
$$\mathtt{out}(i, x_i^0, r_i^0, s, \tau, \mathcal{M}_{\cdot,i}^0, \mathcal{N}_{\cdot,i}^0) \neq \mathtt{out}(i, x_i^1, r_i^1, s, \tau, \mathcal{M}_{\cdot,i}^1, \mathcal{N}_{\cdot,i}^1)$$
   and both are not $\perp$. *Otherwise* $\mathcal{C}$ *outputs* 0.

We call a protocol (ν, RIR, μ)-transcript non-malleable *if any PPT algorithm* $\mathcal{A}$ *can make* $\mathcal{C}$ *output* 1 *in the above game only with probability negligible in* κ.

As mentioned in Section 2.1 we do not necessary require that `out` depends on x_i, r_i. Thus in practice we use a slightly more general definition where also outputs of parties that are not in RIR are considered.

3.2 Simulating Verifiable Protocols: Input-Aware Simulation

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 for our approach is being able to parameterize the operation of simulators for protocols being compiled, as well as giving them external input on 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 explained in Section 1.1. 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.

Über Simulator $\mathcal{S}^{\mathtt{U}}$: We will now start defining the notion of an *über simulator* for a UC-secure protocol Π realizing a functionality $\mathcal{F}$, which we formally establish in Definition 8. We denote über simulators as $\mathcal{S}^{\mathtt{U}}$, while we denote by $\mathcal{S}$ the original simulator used in the UC proof that Π realizes a (non-NIV) functionality $\mathcal{F}$. Basically, an über simulator $\mathcal{S}^{\mathtt{U}}$ takes the inputs to be used by simulated honest parties in interactions with a copy of the adversary as an external parameter and outputs (through a special tape) the randomness used by these simulated parties. Instead of interacting with an internal copy of the adversary, an über simulator interacts with an *external* copy of the adversary. Moreover, an über simulator allows for responses to queries to simulated functionalities to be given externally. Otherwise $\mathcal{S}^{\mathtt{U}}$ will perform similar actions as a regular simulator, such as extracting inputs of dishonest parties to be sent to $\mathcal{F}$. We remark that most existing simulators for protocols realizing the vast majority of natural UC functionalities can be trivially modified to achieve our notion of über simulator (as we will explain in Section 3.3). Notice that most simulators basically execute the protocol as an honest party would, except that they use random inputs and take advantage of their power over setup functionalities to equivocate the output of the simulated protocol to equal the actual output obtained by executing with certain inputs (held by honest parties). Departing from such a simulator, an über simulator can be constructed by allowing the simulated honest party inputs to be obtained externally, rather than being generated internally.

Syntax of Über Simulator $\mathcal{S}^{\mathtt{U}}$: Let $\mathcal{S}^{\mathtt{U}}$ be a PPT iTM with the same input and output tapes as a regular simulator $\mathcal{S}$ plus additional ones as defined below:

- **Input tapes:** a tape for the input from the environment $\mathcal{Z}$, a tape for messages from an ideal functionality $\mathcal{F}$, a tape for inputs for the simulated honest parties, a tape for messages from an *external* adversary $\mathcal{A}$ and a tape for messages from the global setup ideal functionalities in the hybrid model where Π is defined.
- **Output tapes:** tapes for output to $\mathcal{Z}$, tapes for messages to $\mathcal{F}, \mathcal{A}$, tapes for messages to the ideal functionalities in the hybrid model where Π is defined as well as a special "control output tape" that outputs the randomness used by simulated honest parties.

We furthermore define the following two properties of *simulation- and execution-consistency.* Simulation consistency is straightforward and says that any regularly simulated execution is indistinguishable from an execution with $\mathcal{S}^{\mathtt{U}}$ when operating as $\mathcal{S}$ does (*i.e.* with direct access to a copy of the adversary $\mathcal{A}$, functionality $\mathcal{F}$ and a global setup), 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 6 (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 $\mathcal{S}^{\mathtt{U}}$ directly accessing a copy of $\mathcal{A}$ and 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; it takes as input a uniform randomness tape and a tape for simulated honest party inputs sampled in such a way that these inputs 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 for simulated honest parties output by an über simulator $\mathcal{S}^{\mathtt{U}}$ parameterized with the same inputs as the real honest parties to be consistent with the transcripts of a real protocol execution.

Definition 7 (Execution Consistency). *We say that the PPT iTM $\mathcal{S}^{\mathtt{U}}$ is* Execution-consistent *if no PPT iTM $\mathcal{Z}$ for any PPT iTM $\mathcal{A}$ can distinguish:*

1. *$\mathcal{F} \circ \mathcal{S}^{\mathtt{U}}, (R_{h_1}, \ldots, R_{h_k}) \stackrel{\$}{\leftarrow} \mathcal{S}^{\mathtt{U}}$: An ideal execution $\mathcal{F}$ with $\mathcal{S}^{\mathtt{U}}$ where $\mathcal{S}^{\mathtt{U}}$ is parameterized with simulated honest party inputs $(x_{h_1}, \ldots, x_{h_k})$, interacts with $\mathcal{A}$ and with global setup ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_r$, outputting $(R_{h_1}, \ldots, R_{h_k})$ on its special "control output tape" (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 $(x_{h_1}, R_{h_1}), \ldots, (x_{h_k}, R_{h_k})$ (i.e. identical to those of $\mathcal{S}^{\mathtt{U}}$) with global setup ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_r$.*

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 8 (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 6 and 7 for any PPT environment $\mathcal{Z}$ and adversary $\mathcal{A}$.*

3.3 Input-aware simulation for existing protocols.

As outlined in Section 1.3 it is not necessary for each UC-secure protocol to additionally define an über simulator. We 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 (all 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 `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 $\mathcal{P}_{h_1}, \ldots, \mathcal{P}_{h_k}$ taking input/randomness $(x_{h_1}, R_{h_1}), \ldots, (x_{h_k}, R_{h_k})$ 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 $x_{h_1}, \ldots, x_{h_k}$ for simulated the honest parties on its input tapes, samples uniform randomness $R_{h_1}, \ldots, R_{h_k}$ and runs $\mathtt{GenRand}(1^\kappa, R_{h_1}, \ldots, R_{h_k}, x_{h_1}, ..., x_{h_k})$ 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 $\mathcal{F}_1, \ldots, \mathcal{F}_r$ and $\mathcal{S}$. In the end, $\mathcal{S}^{\mathtt{U}}$ outputs $R_{h_1}, \ldots, R_{h_k}$ 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 $\mathcal{F}_1, \ldots, \mathcal{F}_r$-hybrid model where all $\mathcal{F}_1, \ldots, \mathcal{F}_r$ 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 8, it must both simulation and execution-consistent.

First, we will show that $\mathcal{S}^{\mathtt{U}}$ is simulation-consistent according Definition 6, 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 $\mathcal{F}_1, \ldots, \mathcal{F}_r$ and $\mathcal{A}$) and $\mathcal{S}$ is simply forwarded by $\mathcal{S}^{\mathtt{U}}$ to/from $\mathcal{S}$. Notice that, since all $\mathcal{F}_1, \ldots, \mathcal{F}_r$ are global functionalities, $\mathcal{S}$ 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 $\mathtt{GenRand}(1^\kappa, R_{h_1}, \ldots, R_{h_k}, x_{h_1}, ..., x_{h_k})$ 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 $\mathtt{GenRand}$ by definition guarantees that an execution of $\mathcal{S}$ with randomness tape T obtained from executing $\mathtt{GenRand}(1^\kappa, R_{h_1}, \ldots, R_{h_k}, x_{h_1}, ..., x_{h_k})$ 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. $\square$

3.4 Functionalities $\mathcal{F}^{\mathtt{V}}$ with embedded Über Simulator $\mathcal{S}^{\mathtt{U}}$

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 as defined in Definition 8 that is internally executed by the functionality wrapper $\mathcal{F}^{\mathtt{V}}$ presented in Figure 3, which can be accessed by an ideal adversary (*i.e.* $\mathcal{F}^{\mathtt{V}}$'s Simulator) interacting with $\mathcal{F}^{\mathtt{V}}$ through interfaces $\mathbf{Input}_i$ and $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$. Moreover, $\mathcal{F}^{\mathtt{V}}$ allows $\mathcal{S}^{\mathtt{U}}$ to query global setup functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_n$ on behalf of honest parties.

The basic idea is that the internal copy of $\mathcal{S}^{\mathtt{U}}$ executed by $\mathcal{F}^{\mathtt{V}}$ will take care of simulating the original protocol Π that realizes $\mathcal{F}$ being compiled into a publicly verifiable protocol $\Pi^{\mathtt{V}}$ that realizes $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$, while the external $\mathcal{S}$ interacting with $\mathcal{F}^{\mathtt{V}}$ will take care of simulating the additional protocol steps and building blocks used in obtaining public verifiability in $\Pi^{\mathtt{V}}$. In order to do so, $\mathcal{F}^{\mathtt{V}}$ will parameterize $\mathcal{S}^{\mathtt{U}}$ with the inputs of all honest parties $\mathcal{P}_i$, which are received through interface $\mathbf{Input}_i$. As the execution progresses, $\mathcal{S}$ executes the compiled protocol $\Pi^{\mathtt{V}}$ (presented in Figures 6 and 7) with an internal copy $\mathcal{A}$ of the adversary and extracts the messages of the original protocol Π being compiled from this execution, forwarding these messages to $\mathcal{S}^{\mathtt{U}}$ through the interface $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$. Moreover, $\mathcal{S}$ will provide answers to queries to setup functionalities from $\mathcal{A}$ as instructed by $\mathcal{S}^{\mathtt{U}}$ also through interface $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$. All the while, queries from honest parties simulated by $\mathcal{S}^{\mathtt{U}}$ to setup functionalities are directly forwarded back and forth by $\mathcal{F}^{\mathtt{V}}$. If verification is ever activated by an honest party $\mathcal{P}_i$ (and $\mathcal{P}_i \in \mathsf{RIR}$), $\mathcal{F}^{\mathtt{V}}$ not only leaks that party's input to $\mathcal{S}$ but also leaks that party's randomness R_{h_i} in the simulated execution with $\mathcal{S}^{\mathtt{U}}$ (provided by $\mathcal{S}^{\mathtt{U}}$). As we discuss in Section 5, this will allow $\mathcal{S}$ to simulate verification, since it now has both a valid transcript of an execution of $\Pi^{\mathtt{V}}$ with $\mathcal{A}$ and a matching input and randomness pair that matches that transcript (provided by $\mathcal{F}^{\mathtt{V}}$ with the help of $\mathcal{S}^{\mathtt{U}}$).

We remark that this strategy does not give the simulator $\mathcal{S}$ any extra power in simulating an execution of the compiled protocol $\Pi^{\mathtt{V}}$ towards $\mathcal{A}$ other than the power the simulator for the original protocol Π already has. Notice that the access to $\mathcal{S}^{\mathtt{U}}$ given by $\mathcal{F}^{\mathtt{V}}$ to $\mathcal{S}$ does not allow it to obtain any information about the inputs of honest parties, since an execution with $\mathcal{S}^{\mathtt{U}}$ parameterized by these inputs is indistinguishable from an execution with $\mathcal{S}^{\mathtt{U}}$ parameterized by random inputs (as is the case with the original simulator) according to Definition 8.

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

$\mathcal{F}_{\mathsf{PJAuth}}$ interacts with a set of authenticating parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, 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 $k < n$. $\mathcal{F}_{\mathsf{PJAuth}}$ has an initially empty list $\mathcal{L}$.

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 $(sid, ssid, \mathcal{P}_i, m, \sigma_1, \ldots, \sigma_n)$ to $\mathcal{L}$.

Public Verification: Upon receiving $(\textsc{Verify}, sid, ssid, \mathcal{P}_i, m, \sigma_1, \ldots, \sigma_n)$ from a party $\mathcal{V}_i \in \mathcal{V}$, if $(sid, ssid, \mathcal{P}_i, m, \sigma_1, \ldots, \sigma_n) \in \mathcal{L}$, 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 Joint 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 compiler later 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 5 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).

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.

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 $\mathcal{P}_{\mathtt{snd}}$ that is allowed to input messages to be jointly authenticated; a set of parties $\mathcal{P}$ who can read input messages given by $\mathcal{P}_{\mathtt{snd}}$ and jointly authenticate them; a set of bureaucrats $\mathcal{B}$ who do not see the message but jointly authenticate that $\mathcal{P}_{\mathtt{snd}}$ has sent a certain (still unknown) committed message to the parties $\mathcal{P}$. Notice that

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

$\mathcal{F}_{\mathsf{PSAuth}}$ interacts with a special party $\mathcal{P}_{\mathsf{snd}}$, a set of authenticating parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$, a set of bureaucrats $\mathcal{B} = \{\mathcal{B}_1, \dots, \mathcal{B}_b\}$, 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 $\mathcal{L}$.

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

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

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

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

$\mathcal{F}_{\mathsf{PSAuth}}$ does not aid in delivering the message input by $\mathcal{P}_{\mathsf{snd}}$ 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 $\mathcal{P}_{\mathsf{snd}}$, while the responsibility for notifying any other party that plaintext verification is possible lies with $\mathcal{P}_{\mathsf{snd}}$ 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 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 Supplementary Material C.

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 Supplementary Material D we present functionality $\mathcal{F}_{\mathsf{IRAuth}}$ that implements this. The functionality works in the sense of cir of Definition 5, 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, 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 $\mathcal{P}_{\mathtt{snd}}$, 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 5 needs rather strong additional tools such as the authentication functionalities from the previous section. In this work we focus on protocols according to Definition 5 and as such there are 8 different combinations of parameters (ν, μ) for (ν, RIR, μ)-transcript non-malleable 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 (ν, RIR, μ)-transcript non-malleable 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, RIR, μ)-transcript non-malleable 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 $\mathcal{S}^{\mathtt{U}}$ and do the following:

1. We describe how to construct a protocol $\Pi^{\mathtt{V}}$ 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 3.4 in the (global) $\mathcal{F}_1^{\mathsf{V}}, \ldots, \mathcal{F}_r^{\mathsf{V}}$-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 formalize 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 $(\mathsf{cir}, \mathsf{RIR}, \mathsf{cmes})$-transcript non-malleable version of any protocol that is (ν, RIR, μ)-transcript non-malleable. This is the main step in obtaining a publicly verifiable version of an originally (ν, RIR, μ)-transcript non-malleable 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 *does 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, RIR, μ)-transcript non-malleable 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 $\mathcal{F}_{\mathsf{Sig}}$ and 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 $\sigma_{j,i}^{(\tau,\rho)}$. 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 $\mathcal{F}_1^{\mathsf{V}}, \ldots, \mathcal{F}_r^{\mathsf{V}}$ that have the same interfaces as defined in Definition 3. If we intend to achieve public verifiability then each such $\mathcal{F}_q^{\mathsf{V}}$ must also be publicly verifiable, whereas in the case of private verifiability either type of functionality is fine. For any such $\mathcal{F}_q^{\mathsf{V}}$ we can then establish if a certain message $\mathtt{mres}_{q,i}$ was indeed sent to $\mathcal{P}_i$ or not. If $\mathcal{F}_q^{\mathsf{V}}$ 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.

5.2 Public Verifiability Compiler

We now show how to formally embed the aforementioned transformations into a protocol in order to achieve non-interactive UC verifiability. The basic idea of this construction is to turn any $(\mathsf{cir}, \mathsf{RIR}, \mu)$-transcript non-malleable protocol into a $(\mathsf{cir}, \mathsf{RIR}, \mathsf{cmes})$-transcript non-malleable protocol by forcing the adversary to commit to all the corrupted parties' randomness, inputs and messages. While this might be overkill for some protocols, we focus on the worst case scenario of compiling $(\mathsf{cir}, \mathsf{RIR}, \mathsf{ncmes})$-transcript non-malleable protocols, since it is the most challenging. Note that, after making a protocol $(\mathsf{cir}, \mathsf{RIR}, \mathsf{cmes})$-transcript non-malleable, 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 bulletin 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 $(\mathsf{cir}, \mathsf{RIR}, \mu)$-transcript non-malleable protocol we use as the starting point a $(\mathsf{cir}, \mathsf{RIR}, \mathsf{cmes})$-transcript non-malleable protocol or at least reduced if we depart from another protocol where some of the messages are naturally fixed (*e.g.* a $(\mathsf{cir}, \mathsf{RIR}, \mathsf{chmes})$-transcript non-malleable protocol).

Given a (ν, RIR, μ)-transcript non-malleable protocol $\Pi = (\mathtt{nmes}, \mathtt{out})$ that UC realizes an ideal functionality $\mathcal{F}$ in the (global) $\mathcal{F}_1, \ldots, \mathcal{F}_r$-hybrid model with über simulator $\mathcal{S}^{\mathtt{U}}$, 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 6 and 7.

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

$\Pi^{\mathtt{V}}$ is parameterized by a protocol Π with next message function $\mathtt{nmes}$ and output function $\mathtt{out}$ as defined in Section 2.1. $\Pi^{\mathtt{V}}$ 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 Π has used possibly non-verifiable versions thereof. All of these functionalities are available to the verifiers $\mathcal{V}$. Set up one copy $\mathcal{F}_{\mathsf{PSAuth}}^{(i,j)}$ for any private communication where $\mathcal{P}_i$ is $\mathcal{P}_{\mathsf{snd}}$, $\mathcal{P}_j$ acts as authenticating party and all other parties $\mathcal{P} \setminus \{\mathcal{P}_i, \mathcal{P}_j\}$ are 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 $\mathcal{P}_i$ sends $(\textsc{Input}, sid, \mathcal{P}_i, (x_i, r_i))$ to $\mathcal{F}_{\mathsf{IRAuth}}$, while each $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$ sends $(\textsc{Blind-Auth}, sid, \mathcal{P}_i, \mathcal{P}_j)$. Afterwards, $\mathcal{P}_i$ runs Π.**Input**$_i$.

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

Output$_i^{(\tau)}$: $\mathcal{P}_i$ does the same as in Π.**Output**$_i^{(\tau)}$.

Register Verifier: $\mathcal{V}$ sends $(\textsc{Register}, sid)$ to each $\mathcal{F}_q^{\mathtt{V}}$ for $q \in [r]$.

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

Fig. 6. The protocol $\Pi^{\mathtt{V}}$ which makes the $(\mathsf{cir}, \mathsf{RIR}, \mu)$-transcript non-malleable protocol Π publicly verifiable.

Protocol Π^{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 (Cannot-Verify, sid, j).
   For each functionality $\mathcal{F}_q^{\mathsf{V}}$ verify that $\mathcal{N}_{\cdot,i}$ is valid by doing the following:
   - If $\mathcal{F}_q^{\mathsf{V}}$ is Input-Private then send (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 $\mathcal{F}_q^{\mathsf{V}}$ returns (Verify, sid, j, B) with $B \neq \emptyset$ or (Cannot-Verify, sid, j) then output (Cannot-Verify, sid, j).
   - If $\mathcal{F}_q^{\mathsf{V}}$ is Input-Revealing then instead send (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 $\mathcal{F}_q^{\mathsf{V}}$ returns (Verify, sid, j, f, B) with $B \neq \emptyset$, $f = 0$ or if it returns (Cannot-Verify, sid, j) then output (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 (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 (Verify, sid, k, f, B).

Fig. 7. A protocol Π^{V} that makes the ($\mathsf{cir}, \mathsf{RIR}, \mathsf{cmes}$)-transcript non-malleable protocol Π publicly verifiable (continuation).

Theorem 1. *Let Π be a ($\mathsf{cir}, \mathsf{RIR}, \mu$)-transcript non-malleable protocol that UC realizes an ideal functionality $\mathcal{F}$ in the (global) $\mathcal{F}_1^{\mathsf{V}}, \ldots, \mathcal{F}_r^{\mathsf{V}}$-hybrid model with über simulator $\mathcal{S}^{\mathsf{U}}$. Then Π^{V} UC-realizes the publicly verifiable ideal functionality $\mathcal{F}^{\mathsf{V}}[\mathcal{F}]$ in the $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{IRAuth}}, \mathcal{F}_{\mathsf{BB}}, \mathcal{F}_1^{\mathsf{V}}, \ldots, \mathcal{F}_r^{\mathsf{V}}$-hybrid model.*

Proof. In order to prove Theorem 1 we construct a simulator $\mathcal{S}$ that interacts with environment $\mathcal{Z}$, functionality $\mathcal{F}^{\mathsf{V}}[\mathcal{F}]$, global functionalities $\mathcal{F}_1^{\mathsf{V}}, \ldots, \mathcal{F}_r^{\mathsf{V}}$ 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 Π^{V}. $\mathcal{S}$ forwards all communication between $\mathcal{A}$ and $\mathcal{Z}$. $\mathcal{S}$ simulates setup functionalities $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{IRAuth}}, \mathcal{F}_{\mathsf{BB}}, \mathcal{F}_1^{\mathsf{V}}, \ldots, \mathcal{F}_r^{\mathsf{V}}$ 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 $\mathcal{F}^{\mathsf{V}}[\mathcal{F}]$ (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 $\mathcal{S}^{\mathsf{U}}$ incorporated into $\mathcal{F}^{\mathsf{V}}[\mathcal{F}]$. In the Input phase of Π^{V}, $\mathcal{S}$ sends a message the $\mathbf{NMF}_{\mathcal{S}^{\mathsf{U}}}$ interface of $\mathcal{F}^{\mathsf{V}}[\mathcal{F}]$ with any messages received from $\mathcal{A}$. $\mathcal{S}$ forwards to $\mathcal{A}$ any messages returned by $\mathcal{S}^{\mathsf{U}}$ through $\mathbf{NMF}_{\mathcal{S}^{\mathsf{U}}}$ (*i.e.* the messages of simulated honest parties) by simulating messages being sent from honest parties directly to $\mathcal{A}$ and, if necessary, simulating messages sent through $\mathcal{F}_{\mathsf{PSAuth}}, \mathcal{F}_{\mathsf{PJAuth}}, \mathcal{F}_{\mathsf{BB}}$. This allows $\mathcal{S}^{\mathtt{U}}$ to extract $\mathcal{A}$'s inputs and forward them to $\mathcal{F}$ inside the wrapper $\mathcal{F}^{\mathtt{V}}[\mathcal{F}]$. 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 $\mathbf{NMF}_{\mathcal{S}^{\mathtt{U}}}$ 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 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 $\mathcal{F}_{\mathsf{BB}}$ and simulating $\mathcal{F}_{\mathsf{IRAuth}}$ authenticating the opening to (r_i, x_i), 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 $\mathbf{Verify}_j$ 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. $\mathcal{S}$ 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 8. 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 8 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 $(\mathsf{cir}, \mathsf{RIR}, \mathsf{cmes})$-transcript non-malleable protocol in the game of Definition 5 (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 5. Hence, when $\mathcal{S}$ executes the verification phase by following the steps of $\Pi^{\mathtt{V}}$, it is guaranteed by Definition 5 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 Π^{V} and those checks detect cheating correctly with all but negligible probability (by Definition 5), 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. Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 681–698. Springer, Heidelberg, December 2012.
4. Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In Michel Abdalla and Roberto De Prisco, editors, *SCN 14*, volume 8642 of *LNCS*, pages 175–196. Springer, Heidelberg, September 2014.
5. Carsten Baum, Bernardo David, and Rafael Dowsley. Insured mpc: Efficient secure computation with financial penalties. Financial Cryptography and Data Security (FC) 2020. Full version available at `https://eprint.iacr.org/2018/942`.
6. Carsten Baum, Emmanuela Orsini, and Peter Scholl. Efficient secure multiparty computation with identifiable abort. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 461–490. Springer, Heidelberg, October / November 2016.
7. Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Efficient constant-round MPC with identifiable abort and public verifiability, 2020.
8. 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.
9. Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
10. 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.
11. 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.
12. 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.
13. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. Ignacio Cascudo and Bernardo David. SCRAPE: Scalable randomness attested by public entities. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 537–556. Springer, Heidelberg, July 2017.
21. Ignacio Cascudo and Bernardo David. ALBATROSS: publicly attestable batched randomness based on secret sharing. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.
22. Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.
23. Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*, pages 364–369. ACM Press, May 1986.
24. 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.
25. 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.
26. Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018.
27. Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015.

28. 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.
29. Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, August 2017.
30. 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.
31. 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.
32. Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 25(4):680–722, October 2012.
33. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
34. 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.
35. 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.
36. Berry Schoenmakers and Meilof Veeningen. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15*, volume 9092 of *LNCS*, pages 3–22. Springer, Heidelberg, June 2015.

Supplementary Material

A Auxiliary Functionalities

Digital Signatures Ideal Functionality $\mathcal{F}_{\mathsf{Sig}}$. The standard digital signature functionality $\mathcal{F}_{\mathsf{Sig}}$ from [14] is presented in Figure 8. It is also shown in [14] 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 [16] is presented in Figure 9. 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.

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 (KEYGEN, sid) from $\mathcal{P}_s$, verify that $sid = (\mathcal{P}_s, sid')$ for some sid'. If not, ignore the request. Else, hand (KEYGEN, sid) to the adversary $\mathcal{S}$. Upon receiving $(\text{VERIFICATION KEY}, sid, \mathsf{SIG}.vk)$ from $\mathcal{S}$, output $(\text{VERIFICATION KEY}, sid, \mathsf{SIG}.vk)$ to $\mathcal{P}_s$, and record the pair $(\mathcal{P}_s, \mathsf{SIG}.vk)$.

Signature Generation: Upon receiving a message (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 (SIGN, sid, m) to $\mathcal{S}$. Upon receiving $(\text{SIGNATURE}, sid, m, \sigma)$ 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 $(\text{SIGNATURE}, sid, m, \sigma)$ to $\mathcal{P}_s$, and record the entry $(m, \sigma, \mathsf{SIG}.vk, 1)$.

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

Fig. 8. Functionality $\mathcal{F}_{\mathsf{Sig}}$ for Digital Signatures.

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 $(\text{REGISTER}, sid, \mathsf{pk})$ from a party $\mathcal{P}_i \in \mathcal{P}$, send $(\text{REGISTERING}, sid, \mathsf{pk})$ 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 $(\text{RETRIEVE}, sid, \mathcal{P}_j)$ from a party $\mathcal{P}_i \in \mathcal{P}$, send message $(\text{RETRIEVE}, sid, \mathcal{P}_j)$ to $\mathcal{S}$ and wait for it to return a message $(\text{RETRIEVE}, sid, ok)$. Then, if there is a recorded pair $(\mathcal{P}_j, \mathsf{pk})$ output $(\text{RETRIEVE}, sid, \mathcal{P}_j, \mathsf{pk})$ to $\mathcal{P}_i$. Otherwise, if there is no recorded tuple, return $(\text{RETRIEVE}, sid, \mathcal{P}_j, \perp)$.

Fig. 9. Functionality $\mathcal{F}_{\mathsf{Reg}}$ for Key Registration.

Bulletin Board Ideal Functionality $\mathcal{F}_{\mathsf{BB}}$. In Figure 10 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.

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 $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ 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. 10. Functionality $\mathcal{F}_{\mathsf{BB}}$ for an authenticated Bulletin Board.

B UC Secure Public-Key Encryption and Constructions

It is well-known that the standard public-key encryption functionality $\mathcal{F}_{\mathsf{PKE}}$ from [13,16] can be UC-realized by any IND-CCA secure public-key encryption scheme. One of the main building blocks we use 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. 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 [35,24].

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 $(\mathsf{pk}, \mathsf{sk}) \stackrel{\$}{\leftarrow} \mathsf{KG}(1^\kappa)$, any $\mathsf{m} \in \mathcal{M}$, and $\mathsf{ct} \stackrel{\$}{\leftarrow} \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$, 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 [13,16].

The Pointcheval [35] 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 [35], 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 9 (Partially Trapdoor One-Way Function [35]). *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 [35], which is presented in Definition 10.

Definition 10 (Pointcheval [35] 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 [35] IND-CCA secure Cryptosystem. First, notice that this construction can be instantiated in the restricted observable and programmable global random oracle model of [10]. 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 11 (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 12 (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 [35] 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 [35] 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 13. 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 [35] 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 13 (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 [35] IND-CCA secure cryptosystem to add plaintext verification as follows:

Definition 14 (Pointcheval [35] 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 10.*
- $\mathsf{Enc}(\mathsf{pk}, \mathsf{m})$*: Same as in Definition 10 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 10 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 12 with non-negligible probability. Adversary $\mathcal{A}'$ receives pk from the challenger in the game of Definition 12 and then acts as the challenger in the game of Definition 13, 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 12 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 13 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 12 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 [13,18], 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 [16], 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 [13,18] 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 [13,18]: 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 [13,18], but we will show that there exist IND-CCA secure public-key encryption schemes [35,24] that also realize our extended functionality $\mathcal{F}_{\mathsf{PKEPV}}$.

Realizing $\mathcal{F}_{\mathsf{PKEPV}}$ It is known [13,18] 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 [13,18]. 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 [35] with plaintext verification from Section B to realize $\mathcal{F}_{\mathsf{PKEPV}}$ following the same approach as in [13,18]. This generic construction works in the restricted programmable and observable random oracle model [10] and can be instantiated from the CDH assumption.

We realize $\mathcal{F}_{\mathsf{PKEPV}}$ by extending the encryption protocol π_{PKE} of [13,18], 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 (KeyGen, $sid, \mathcal{P}_{\mathsf{own}}$), $\mathcal{P}_{\mathsf{own}}$ executes $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{KG}(1^\kappa)$, records sk and returns pk. Upon receiving a message (Encrypt, $sid, \mathcal{P}_{\mathsf{own}}, \mathbf{e}', m$), any party $\mathcal{P}_i \in \mathcal{P}$ outputs ct where $(\mathsf{ct}, \pi) \xleftarrow{\$} \mathsf{Enc}(\mathbf{e}', m)$ if $m \in M$ (otherwise it outputs an error message). Upon receiving (Decrypt, $sid, \mathcal{P}_{\mathsf{own}}, c$), $\mathcal{P}_{\mathsf{own}}$ outputs m where $(m, \pi) \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$. Upon receiving a message (Verify, $sid, \mathcal{P}_{\mathsf{own}}, c, m, \pi$), 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 14 can be used to instantiate $\mathit{\Pi}_{\mathsf{PKEPV}}$ 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 14. Protocol* $\mathit{\Pi}_{\mathsf{PKEPV}}$ *instantiated with* PKE *UC realizes* $\mathcal{F}_{\mathsf{PKEPV}}$ *in the restricted programmable and observable random oracle model [10].*

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 $\mathit{\Pi}_{\mathsf{PKEPV}}$ with any adversary $\mathcal{A}$ and dummy parties. Notice that the steps of $\mathit{\Pi}_{\mathsf{PKEPV}}$ dealing

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

$\mathcal{F}_{\mathsf{PKEPV}}$ interacts with a special decrypting party $\mathcal{P}_{\mathtt{own}}$, 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 $(\text{KeyGen}, sid, \mathcal{P}_{\mathtt{own}})$ from a party $\mathcal{P}_{\mathtt{own}} \in \mathcal{P}$ (or $\mathcal{S}$), proceed as follows:
1. Send $(\text{KeyGen}, sid, \mathcal{P}_{\mathtt{own}})$ to $\mathcal{S}$.
2. Receive a value $\mathbf{e}$ from $\mathcal{S}$.
3. Record $\mathbf{e}$ and output $\mathbf{e}$ to $\mathcal{P}_{\mathtt{own}}$.

Encryption: Upon receiving a message $(\text{Encrypt}, sid, \mathcal{P}_{\mathtt{own}}, \mathbf{e}', m)$ 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 $\mathcal{P}_{\mathtt{own}}$ 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 $(\text{Decrypt}, sid, \mathcal{P}_{\mathtt{own}}, c)$ from $\mathcal{P}_{\mathtt{own}}$, 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 $\mathcal{P}_{\mathtt{own}}$. (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 $\mathcal{P}_{\mathtt{own}}$).
2. Otherwise, compute $(m, \pi) \leftarrow D(c)$ and hand (m, π) to $\mathcal{P}_{\mathtt{own}}$.

Plaintext Verification: Upon receiving a message $(\text{Verify}, sid, \mathcal{P}_{\mathtt{own}}, c, m, \pi)$ 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. 11. Public-Key Encryption Functionality with Plaintext Verification $\mathcal{F}_{\mathsf{PKEPV}}$.

with messages $(\text{KeyGen}, sid, \mathcal{P}_{\mathtt{own}})$, $(\text{Encrypt}, sid, \mathcal{P}_{\mathtt{own}}, \mathbf{e}', m)$ and $(\text{Decrypt}, sid, \mathcal{P}_{\mathtt{own}}, c)$ correspond exactly to the protocol of [13,18] realizing the standard public-key encryption, and our simulator can function exactly as the simulator of [13,18]. In fact, all the simulator does is executing $\mathsf{KG}(1^\kappa)$ and setting pk. It is proven in [13,18] 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 $(\text{Verify}, sid, \mathcal{P}_{\mathtt{own}}, c, m, \pi)$, 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 $\mathcal{P}_{\mathtt{own}}$, 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 $(\textsc{Encrypt}, sid, \mathcal{P}_{\mathtt{own}}, m)$ 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 $\mathcal{P}_{\mathtt{own}}$ 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 $(\textsc{Decrypt}, sid, \mathcal{P}_{\mathtt{own}}, c)$ from $\mathcal{P}_{\mathtt{own}}$, proceed as follows (if the input is from another party then ignore):

1. If this is the first decryption request made by $\mathcal{P}_{\mathtt{own}}$ 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 $\mathcal{P}_{\mathtt{own}}$. (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 $\mathcal{P}_{\mathtt{own}}$).
3. Otherwise, compute $(m, \pi) \leftarrow D(c)$ and hand (m, π) to $\mathcal{P}_{\mathtt{own}}$.

Plaintext Verification: Upon receiving a message $(\textsc{Verify}, sid, \mathcal{P}_{\mathtt{own}}, c, m, \pi)$ 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. 12. 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 [16]. 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 be received by an specific party. In order to realize such a functionality, a key

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 $\mathcal{P}_{\mathtt{own}}$, 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 $\mathcal{P}_{\mathtt{own}}$ sends message $(\textsc{KeyGen}, sid, \mathcal{P}_{\mathtt{own}})$ to $\mathcal{F}_{\mathsf{PKEPV}}$, receiving pk. Next, $\mathcal{P}_{\mathtt{own}}$ sends $(\textsc{register}, sid, \mathsf{pk})$ to $\mathcal{F}_{\mathsf{Reg}}$.

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

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

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

Plaintext Verification: Upon receiving a message $(\textsc{Verify}, sid, \mathcal{P}_{\mathtt{own}}, c, m, \pi)$, 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 $(\textsc{Retrieve}, sid, \mathcal{P}_{\mathtt{own}})$ to $\mathcal{F}_{\mathsf{Reg}}$, receiving $(\textsc{Retrieve}, sid, \mathcal{P}_{\mathtt{own}}, \mathsf{pk})$ 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. 13. Protocol π_{CPKEPV} realizing $\mathcal{F}_{\mathsf{CPKEPV}}$.

registration ideal functionality $\mathcal{F}_{\mathsf{Reg}}$ that allows parties to register their public-keys is required. It was shown in [16] 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 [16] 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 12. 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 [16] 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 [16] and follows the same approach for implementing the Plaintext Verification interface. Protocol π_{CPKEPV} realizing $\mathcal{F}_{\mathsf{CPKEPV}}$ is presented in Figure 13.

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 [16]: $\mathcal{S}$ 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 $\mathcal{P}_{\mathtt{own}}$ 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 [16]. 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 $\mathcal{P}_{\mathtt{own}}$) and initializing an instance of $\mathcal{F}_{\mathsf{Sig}}$ for $\mathcal{P}_{\mathtt{snd}}$, each party $\mathcal{P}_i \in \mathcal{P}$ and each bureaucrat $\mathcal{B}_i \in \mathcal{B}$. Next, $\mathcal{P}_{\mathtt{snd}}$, 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 $\mathcal{P}_{\mathtt{snd}}$ 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 $\mathcal{P}_{\mathtt{snd}}$'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 $\mathcal{P}_{\mathtt{snd}}$ or a party $\mathcal{P}_i \in \mathcal{P}$ want to prove a certain message was sent by $\mathcal{P}_{\mathtt{snd}}$ 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 $\mathcal{P}_{\mathtt{snd}}$, 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 14, 15.

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 $\mathcal{P}_{\mathsf{snd}}$, a set of authenticating parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, a set of bureaucrats $\mathcal{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_b\}$ 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 $\mathcal{P}_{\mathsf{own}}$ (*e.g.* $\mathcal{P}_{\mathsf{own}} = \mathcal{P}$[a]. For each party in $\mathcal{P}$, bureaucrat in $\mathcal{B}$ and party $\mathcal{P}_{\mathsf{snd}}$, 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 $\mathcal{P}_{\mathsf{snd}}$, 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 $\mathcal{P}_{\mathsf{snd}}$ register their signing keys by sending $(\textsc{register}, sid, \mathsf{pk})$ to $\mathcal{F}_{\mathsf{Reg}}$.

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

Joint Authentication: Upon receiving a message $(\textsc{Auth}, sid, ssid, \mathcal{P}_{\mathsf{snd}}, m)$, a party $\mathcal{P}_i \in \mathcal{P}$ checks if it has received $(sid, ssid, c, \sigma_{\mathsf{snd}})$ 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_{snd} is a valid signature on c under $\mathcal{P}_{\mathsf{snd}}$'s public-key (retrieved) from $\mathcal{F}_{\mathsf{Reg}}$ according to $\mathcal{P}_{\mathsf{snd}}$'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 $(\textsc{Verify}, sid, \mathcal{P}_{\mathsf{own}}, c, m, \pi)$ to $\mathcal{F}_{\mathsf{CPKEPV}}$ and check that 1 is received as response.
2. Send $(\textsc{Retrieve}, sid, \mathcal{P}_{\mathsf{snd}})$ to $\mathcal{F}_{\mathsf{Reg}}$, wait for $(\textsc{Retrieve}, sid, \mathcal{P}_{\mathsf{snd}}, \mathsf{pk})$, send $(\textsc{verify}, sid, m, \sigma_{\mathsf{snd}}, \mathsf{pk})$ 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 $(\textsc{Blind-Auth}, sid, ssid, \mathcal{P}_{\mathsf{snd}})$, 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 $\mathcal{P}_{\mathsf{own}}$ denote a set of parties instead of single party in $\mathcal{F}_{\mathsf{CPKEPV}}$

Fig. 14. 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 at least one party $\mathcal{P}_i \in \mathcal{P}$ and/or $\mathcal{P}_{\mathsf{snd}}$, acting this way allows $\mathcal{S}$ to perfectly simulate an execution of π_{PSAuth} towards corrupted bureaucrats. Notice

Protocol π_{PSAuth} (Public Verification)

Public Verification: Upon receiving $(\textsc{Verify}, sid, ssid, \mathcal{P}_{\mathsf{snd}}, m, \sigma_{\mathsf{snd}}, \sigma_1, \ldots, \sigma_n, \hat{\sigma}_1, \ldots, \hat{\sigma}_b)$, a party $\mathcal{V}_i \in \mathcal{V}$ first parses all tokens $\sigma_{\mathsf{snd}}, \sigma_1, \ldots, \sigma_n$ 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 $\mathcal{P}_{\mathsf{snd}}$, 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 $(\textsc{Verify}, sid, ssid, \mathcal{P}_{\mathsf{snd}}, m, v)$.

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

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

Functionality $\mathcal{F}_{\mathsf{IRAuth}}$ is described in Figure 16

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 $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$, 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. 16. Input and Randomness Authentication Functionality $\mathcal{F}_{\mathsf{IRAuth}}$.