






(Public) Verifiability for Composable Protocols Without Adaptivity or Zero-Knowledge

Carsten Baum¹ , Bernardo David² , and Rafael Dowsley³ 

¹ Aarhus University, Aarhus, Denmark
cbaum@cs.au.dk

² IT University Copenhagen, Copenhagen, Denmark
beda@itu.dk

³ Monash University, Melbourne, Australia
rafael.dowsley@monash.edu

Abstract. The Universal Composability (UC) framework (FOCS '01) is the current standard for proving security of cryptographic protocols under composition. It allows to reason about complex protocol structures in a bottom-up fashion: any building block that is UC-secure can be composed arbitrarily with any other UC-secure construction while retaining their security guarantees. Unfortunately, some protocol properties such as the verifiability of outputs require excessively strong tools to achieve in UC. In particular, “obviously secure” constructions cannot directly be shown to be UC-secure, and verifiability of building blocks does not easily carry over to verifiability of the composed construction. In this work, we study Non-Interactive (Public) Verifiability of UC protocols, *i.e.* under which conditions a verifier can ascertain that a party obtained a specific output from the protocol. The verifier may have been part of the protocol execution or not, as in the case of public verifiability. We consider a setting used in a number of applications where it is ok to reveal the input of the party whose output gets verified and analyze under which conditions such verifiability can generically be achieved using “cheap” cryptographic primitives. That is, we avoid having to rely on 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 provably composable security and efficiency in mind.

1 Introduction

Universal Composability (UC) [14] is currently the most popular framework for designing and proving security of cryptographic protocols under arbitrary com-

Funded by the European Research Council (ERC) under the European Unions’ Horizon 2020 program under grant agreement No 669255 (MPCPRO).

Supported by the Concordium Foundation and by the Independent Research Fund Denmark grants number 9040-00399B (TrA²C), number 9131-00075B (PUMA) and number 0165-00079B (Foundations of Privacy Preserving and Accountable Decentralized Protocols).

position. 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 protocol security is argued on a very high level only. This is partially due to the complexity of fully expressing (and then proving) a protocol in UC, but also because achieving provable (UC) security sometimes requires additional, seemingly unnecessary protocol steps or assumptions.

One such case is that of (public) verifiability, which is the focus of this work. A verifiable protocol allows each protocol participant \mathcal{P}_i to check if another party \mathcal{P}_j in the end of the protocol obtained a certain claimed output (or that it aborted). A publicly verifiable protocol has this property even for external verifiers that did not take part in the protocol itself. Public verifiability is particularly important in the setting of decentralized systems and public ledgers (*e.g.* blockchains [24, 27, 28, 32, 35]), where new parties can join an ongoing protocol execution on-the-fly after verifying that their view of the protocol is valid. It also plays a central role in a recent line of research [2, 5, 11, 33] on secure multi-party computation (MPC) protocols that rely on a public ledger to achieve fairness (*i.e.* ensuring either all parties obtain the protocol output or nobody does, including the adversary) by penalizing cheating parties, circumventing fundamental impossibility results [25]. Protocol verifiability also finds applications in MPC protocols that have identifiable abort such as [8, 9, 30], where all parties in the protocol either agree on the output or agree on the set of cheaters. Furthermore, public verifiability is an intrinsic property of randomness beacons [22, 23] and a central component of provably secure Proof-of-Stake blockchain protocols [5, 24, 33]. 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.

1.1 The Problems of Achieving (Public) Verifiability in UC

Consider a UC functionality \mathcal{F} which has one round of inputs by the parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$, computes outputs based on the inputs and in the end sends these outputs to each \mathcal{P}_i . In this work, we are interested in adding verifiability to \mathcal{F} to obtain an extended functionality \mathcal{F}^V . This functionality \mathcal{F}^V performs the same operations as \mathcal{F} , but it additionally allows verifiers to confirm that certain inputs were provided by a party \mathcal{P}_i to \mathcal{F}^V to perform these operations and that certain outputs of these operations were given to \mathcal{P}_i from \mathcal{F}^V . Moreover, we want to obtain a protocol Π^V realizing \mathcal{F}^V from an existing protocol Π that realized \mathcal{F} . More concretely, we are interested in compiling a UC-secure protocol Π into a UC-secure counterpart Π^V that has (public) verifiability.

The (intuitive) first step is to construct Π^V where each party commits to its inputs and randomness. The parties then run Π using the committed input and randomness, exchanging exchange authenticated messages. This approach assumes that we are okay with revealing the inputs after Π is completed in case cheating is suspected, which we will discuss in more detail. Intuitively, this

yields a simple verification procedure: each party can use the committed inputs and randomness of all other parties to re-execute Π in its head and compare the resulting messages to the authenticated protocol transcript. Any external verifier could do the same based on the commitments and an authenticated transcript of Π . Unfortunately, using this simple approach leads to adaptivity problems when trying to prove Π^V secure: in the security proof, the simulation must have been performed without knowing the actual inputs to the functionality. But afterwards, these inputs become known to the verifier so the simulator must be able to explain the whole transcript of Π in terms of the previously unknown inputs, requiring adaptive security of Π to begin with. Similar issues have been observed before (e.g. [30]). This means that any such Π^V would be quite inefficient, since adaptive protocols Π are often significantly less efficient than their counterparts with static security.

Consider, as an example, a 2PC protocol Π_{2PC} with active security based on Garbled Circuits (GCs) such as [18, 34]. Protocol Π_{2PC} is executed by a sender \mathcal{P}_1 and a receiver \mathcal{P}_2 (where only \mathcal{P}_2 obtains output) as follows: (1) \mathcal{P}_1 generates multiple GCs together with input keys for each circuit. \mathcal{P}_1 commits to the GCs and their input keys. It inputs the input keys belonging to \mathcal{P}_2 into an Oblivious Transfer (OT) functionality \mathcal{F}_{OT} ; (2) \mathcal{P}_2 uses \mathcal{F}_{OT} to obtain its input keys; (3) \mathcal{P}_1 decommits the GCs and its own input keys; (4) \mathcal{P}_2 evaluates the GCs. Both parties run a consistency check showing that most GCs were correctly generated and that their input keys are consistent. The security proof of Π_{2PC} (for static security) usually consists of simulators for a corrupted sender (\mathcal{S}_1) and receiver (\mathcal{S}_2). \mathcal{S}_1 sends random inputs to \mathcal{F}_{OT} , extracts the inputs of \mathcal{P}_1 and then checks that the GCs were generated correctly by the malicious \mathcal{P}_1 . For \mathcal{S}_2 the standard strategy is to first extract the input x_2 of the malicious \mathcal{P}_2 using \mathcal{F}_{OT} , then to obtain the output y from the functionality \mathcal{F}_{2PC} , to choose a random input \tilde{x}_1 and finally to simulate GCs such that they output y for the input keys of \tilde{x}_1, x_2 . In order to make Π_{2PC} verifiable (with respect to revealing inputs and outputs), let \mathcal{F}_{2PC}^V release the real input x_1 of \mathcal{P}_1 after the computation finished. But in \mathcal{S}_2 we generated the GCs such that for the dummy input \tilde{x}_1 it outputs y , so the garbling may not even be a correct garbling of the given circuit. There might not exist randomness to explain the output of \mathcal{S}_2 consistently with x_1 , unless Π_{2PC} was an adaptively secure protocol.

This seems counter-intuitive: beyond the technical reason to allow (UC) simulation of verifiability, we see no explanation why only adaptively secure protocols should be verifiable when following the aforementioned compilation steps.

1.2 Our Contributions

In this work, we show how to compile a large class of statically UC-secure protocols into publicly verifiable versions that allow a party to non-interactively prove that it obtained a certain output by revealing its input. We focus on a setting where at least one party is assumed to be honest, and where the compiled protocol was already maliciously secure to begin with. While revealing an input is a caveat, this flavor of (public) verifiability is sufficient for a number of

applications (e.g. [5, 7, 23]) and allows us to circumvent the need for expensive generic zero knowledge proofs and adaptive security (as needed in [30, 33]). We introduce a compiler relying only on commitments and “joint authentication” functionalities that can be realized with cheap public-key primitives.

Our approach is compatible with protocols realizing non-reactive functionalities such as OT, Commitments 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 and to perform verification. This wrapper is designed to amalgamate the reactive nature of UC with non-interactivity and might be of independent interest. Extending the results to reactive functionalities is an interesting open problem.

When is Revealing Inputs for Verification Justifiable? Although our focus on revealing inputs might seem very restrictive, there is a quite substantial set of protocols where it can be applied. As a starting point, our techniques can be used to instantiate preprocessing for UC-secure MPC with Identifiable Abort without adaptive assumptions [9, 30]. Our approach also applies when one wants to publicly and randomly sample from a distribution and identify cheaters who disturbed the process. For example, our results have already been used as an essential tool in follow-up work constructing UC randomness beacons [23]. A third application is to bootstrap MPC without output verifiability to MPC with output verifiability *without revealing of inputs*. Here, each physical party \mathcal{P}_i in the protocol Π_{MPC} runs two virtual parties $\mathcal{P}_i^C, \mathcal{P}_i^V$. It will give \mathcal{P}_i^C the actual input x (while \mathcal{P}_i^V has no input), and both parties obtain the same output y from Π_{MPC} . Now, in order to convince a verifier that \mathcal{P}_i had y as output, it can “sacrifice” \mathcal{P}_i^V and reveal its randomness for verification. Observe that this requires Π_{MPC} to be secure against a dishonest majority of parties. A fourth application lies in achieving cheater identification in the output phase of MPC protocols, which is a prerequisite for obtaining MPC with monetary fairness such as [2, 5, 11, 33]. For example, using our techniques, it is possible to construct the publicly verifiable building blocks of the output phase of Insured MPC [5] and related applications [7] since the inputs of the output phase with cheater identification are supposed to be revealed anyway. In [5] the authors had to individually redefine each functionality with respect to verifiability and reprove the security of each protocol involved. Using our techniques, we show in the full version [6] that this tedious task can be avoided and that the same result can be obtained by inspecting the primitives used in their protocol and verifying that the protocols fulfill the requirements of our compiler.

Shortcomings of Other Approaches. In case adaptive security is required, it is well-known that adaptively secure protocols usually have larger computation or communication overheads (or stronger assumptions) than their statically secure counterparts. For example, Yao’s Garbling Scheme and optimizations thereof are highly efficient with static security (e.g. [38]) but achieve similar performance with adaptive security only for NC1-circuits [31] (unless one relies on Random Oracles [10]). When implementing Π_{2PC} , one would also additionally

have to realize an adaptively UC-secure \mathcal{F}_{OT} , which is also cheaper with static instead of adaptive security. This is also true when OT-extension is used [20, 21].

Previous works such as [33] obtain public verifiability, even without revealing inputs and without adaptive protocols, by using generic UC-NIZKs. They follow the GMW paradigm [29] where each party would prove in every protocol step of Π that it created all messages correctly, given all previous messages as well as commitments to inputs and randomness. To the best of our knowledge, no work that uses UC-NIZKs to achieve verifiability estimated concrete parameters for their constructions. This is because the UC-NIZKs, in addition to proving the protocol steps, also have to use the code of the cryptographic primitives in a white-box way. That also means that UC-NIZKs cannot be applied if the compiled protocol Π uses Random Oracles, which are popular in efficient protocols.

Another solution for verifiability, which also does not require an adaptive protocol and that works in the case that Π is an MPC protocol, is to i) let Π commit to the output of y_i of each \mathcal{P}_i by running a commit algorithm for a non-interactive commitment scheme inside Π ; ii) output all these commitments to all parties, which sign them and broadcast the signed commitments to each other; and iii) reveal outputs and commitment openings to the respective parties. Obviously, this does not generalize to arbitrary protocols Π , whereas our approach does. Additionally, in this approach one needs to evaluate the commitment algorithm white-box in MPC. Evaluating cryptographic primitives inside MPC can be costly, in particular if the MPC protocol is defined over a ring where the commitment algorithm has a large circuit. This also rules out cheap Random Oracle-based commitments.

Efficiency. The only overheads in relation to the original protocol required by our compiler are a simple commitment (*e.g.* based on a random oracle) on the input/randomness of each party and the subsequent joint authentication of this commitment as well as of subsequent messages. If messages are exchanged over public channels, this joint authentication can be done by requiring each party to compute multisignatures on the messages exchanged in each round and then combining these signatures into a single multisignature, saving on space. If messages are exchanged over private channels, there is an extra overhead of computing 2 modular exponentiations and transmitting a string of security parameter size per message, which is needed for our private joint authentication scheme. The verification procedure requires the verifier to re-execute the protocol on the jointly authenticated transcript of the protocol using a party's opened input/randomness. While this seems expensive, notice that executing the protocol's next message function is strictly cheaper than verifying a NIZK showing that every message in the transcript is correctly computed according to this function, which is required by previous schemes and that would also add the overhead of having each party compute such a NIZK for every message they send.

1.3 Our Techniques

We construct a compiler that generically achieves public verifiability for protocols with one round of input followed by multiple computation and output rounds as formalized in Sect. 2. For this, we start with an observation similar to [30], namely that by fixing the inputs, randomness and messages in a protocol Π 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. Our compiler creates a protocol Π^V that fixes 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. This idea of fixing messages for the purpose of public verifiability is not new, and other works that focus on it such as [3, 33] have taken a similar route. However, fixing all messages exchanged in the original protocol Π is costly and might be overkill for some protocols. We explore this concept in the notion of *transcript non-malleability* as defined in Sect. 3. There, we formalize the intuition that we might not need that all exchanged messages are fixed in some protocols: *e.g.* an adversary that is allowed to replace messages exchanged between dishonest parties possibly does not have enough leverage to forge a consistent transcript for a different output.

Proving Security in UC: It might seem obvious that Π^V , i.e. a version of Π with all of its inputs and messages fixed, is publicly verifiable and implements \mathcal{F}^V . Unfortunately, as we outlined above, a construction of a simulator \mathcal{S}^V in the proof of security needs to assume that Π is adaptively secure. In Sect. 3.1 we address this by using *input-aware simulators* (or *über simulators*) \mathcal{S}^U . These are special simulators which can be parameterized with the inputs for the simulated honest parties, generating transcripts consistent with these inputs but indistinguishable from the transcripts of \mathcal{S} . We then embed an über simulator of a protocol Π into the publicly verifiable functionality \mathcal{F}^V . This delegates the simulation of Π to the internal über simulator of \mathcal{F}^V – whereas in our naive approach, \mathcal{S}^V had to simulate Π itself. Since we let \mathcal{F}^V only release the transcripts that \mathcal{S}^U generates, this does not leak any additional information to the adversary. Moreover, \mathcal{S}^U now also extracts the inputs of the dishonest parties.

Getting Über Simulators (Almost) for Free: Following our example with Π_{2PC} from Sect. 1.1, \mathcal{S}_1 for a corrupted sender uses a random input to \mathcal{F}_{OT} and otherwise follows Π_{2PC} . Towards constructing \mathcal{S}_1^U , observe that as \mathcal{F}_{OT} by its own UC-security hides the input of \mathcal{P}_2 , running \mathcal{S}_1 inside \mathcal{F}_{2PC}^V using real inputs of \mathcal{P}_2 is indistinguishable and we can use such a modified \mathcal{S}_1 as \mathcal{S}_1^U . Conversely, we can also construct \mathcal{S}_2^U , which runs Π_{2PC} based on the input x_1 that it obtains. By the UC-security of Π_{2PC} , the distribution of \mathcal{S}_2^U will be indistinguishable from \mathcal{S}_2 . As can be seen from this example, an efficient über simulator must not be artificial or strong, but could be quite simply obtained from either the existing protocol or existing \mathcal{S} . Its requirement also differs from requiring adaptivity of Π_{2PC} : \mathcal{S}_2^U still only requires Π_{2PC} to be statically secure. In fact, this strategy

for constructing an über simulator works for any protocols that simulate their online phase in the security proof using “artificial” fixed inputs and otherwise run the protocol honestly while they are able to extract inputs (*e.g.* MPC protocols such as [26, 36]). Hence, we can directly make a large class of protocols verifiable. This is discussed further in the full version [6].

How to Realize Transcript Non-malleability. Besides fixing inputs and randomness, in order to construct compilers from Π to Π^V we need to fix the transcript of Π . For this, we have parties in Π^V use what we call “joint authentication” (defined in Sect. 4). Joint Authentication works for both public and private messages. In the public case, joint authentication is achieved by 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 the rightful receivers. Later on, any party who received that private message (*i.e.* the opening of the commitment to the message) can publicly prove that it obtained a certain message that was jointly authenticated by all parties involved in Π^V . More importantly, joint authentication does not perform any communication itself but provides authentication tokens that can be verified in a non-interactive manner. In our example with Π_{2PC} , this means that both $\mathcal{P}_1, \mathcal{P}_2$ initially commit to their inputs and randomness and then sign all exchanged messages (checking that each message is signed by its sender).

Putting Things Together. We use the techniques described above to compile any protocol Π that fits one of our transcript non-malleability definitions and UC-realizes a functionality \mathcal{F} in the $\mathcal{F}_1, \dots, \mathcal{F}_n$ -hybrid model into a protocol Π^V that UC-realizes a publicly verifiable \mathcal{F}^V in the $\mathcal{F}_1^V, \dots, \mathcal{F}_n^V$ -hybrid model (*i.e.* assuming that the setup functionalities can also be made publicly verifiable). Moreover, if a global functionality is used as setup, it must allow all parties to make the same queries and obtain the same answers, so that the verification procedure can be performed. Our compilation technique has two main components: 1. commit to and authenticate each party’s input and randomness pairs of Π (fixing input and randomness pairs); 2. execute Π and use public/secret joint authentication to jointly authenticate each exchanged protocol message (fixing the transcript). These steps achieve two goals: allowing parties to publicly and non-interactively show that they have a certain input/randomness pair and transcript, making Π transcript non-malleable, since the adversary cannot lie about its input, randomness or view of the transcript. In order to realize the public verifiability interface of \mathcal{F}^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 as in Π to verify that a given output is obtained. When proving security of this compiler, we delegate the simulation of the original steps of Π to an über simulator \mathcal{S}^U for Π embedded in \mathcal{F}^V . This guarantees that the transcript of \mathcal{S} ’s simulated execution of Π^V is consistent with honest parties’ inputs if they activate public verification and reveal their input. To compile our example GC protocol, we now combine all of the aforementioned steps and additionally assume that \mathcal{F}_{OT} as well as the commitment-functionality are already verifiable. By the compiler theorem, the

resulting protocol is verifiable according to our definition. In the full version [6] we give a detailed example by easily achieving verifiability in [5].

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 [16], 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 [8,30], verifiability of MPC [4,37], incoercible secure computation [1], secure computation on public ledgers [2,11,33], and improved definitions for widely used primitives [12,13]. Another solution to solve the adaptivity requirement was presented in [9], but their approach only works for functionalities without input. A different notion of verifiability was put forward in publicly verifiable covert 2PC protocols such as [3] and its follow-up works, where parties can show that the other party has cheated. Here, both the 2PC protocol and therefore also the verifiability guarantee only hold against covert adversaries, while we focus on the malicious setting. To the best of our knowledge, no previous work has considered a generic definition of non-interactive public verifiability for malicious adversaries in the UC framework nor a black-box compiler for achieving such a notion *without* requiring adaptive security of the underlying protocol or ZK proof systems.

2 Preliminaries

We denote the security parameter by κ . The set $\{1, \dots, n\}$ is denoted by $[n]$ while we write $[n]_i$ to mean $[n] \setminus \{i\}$. We denote by $\text{negl}(x)$ the set of negligible functions in x and abbreviate *probabilistic polynomial time* as PPT. We write $\{0, 1\}^{\text{poly}(\kappa)}$ to denote a set of bit-strings of polynomial length in κ .

Secure Protocols. A protocol Π run by n parties (which we denote as $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$) consists of the algorithms `nmes`, `out` and additional parameters: the number of parties n , the setup resources $\mathcal{F}_1, \dots, \mathcal{F}_r$, 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 external system parameters $s \in \{0, 1\}^{\text{poly}(\kappa)}$ are fixed for the protocol. Each party \mathcal{P}_i uses their input $x_i \in \mathcal{X}$ as well as randomness $r_i \in \{0, 1\}^{\text{poly}(\kappa)}$ for the actual protocol. Formally, Π is described in Fig. 1 with algorithms `nmes` and `out` defined as follows:

`nmes` is a deterministic polynomial-time (DPT) algorithm which on input the party number i , protocol input $x_i \in \mathcal{X}$, randomness $r_i \in \{0, 1\}^{\text{poly}(\kappa)}$, auxiliary input $s \in \{0, 1\}^{\text{poly}(\kappa)}$, 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 $\{\mathbf{m}_{i,j}^{(\tau,\rho)}\}_{j \in [n]_i}, \{\mathbf{mres}_{i,q}^{(\tau,\rho)}\}_{q \in [r]}$.

out is a DPT algorithm which on input the party number i , the protocol input $x_i \in \mathcal{X}$, randomness $r_i \in \{0, 1\}^{\text{poly}(\kappa)}$, auxiliary input $s \in \{0, 1\}^{\text{poly}(\kappa)}$, output round $\tau \in [G]$, a set of messages $\mathcal{M}_{\cdot, i}$ from parties and $\mathcal{N}_{\cdot, i}$ from resources outputs $y_i^{(\tau)}$ which is either an output value or \perp . The values x_i, r_i might not be necessary in every protocol, so **out** might run without these.

Algorithm **mes** generates two different types of messages: 1. **m**-messages, which are used for communication *among parties*; 2. **mres**-messages, which are exchanged *between a party and a functionality*. Therefore, each **mres**-message consists of an interface (**Input** $_i$, **Compute** $^{(\tau)}$, **Output** $_i^{(\tau)}$) with whom the party wants to communicate as well as the actual payload. Each message that is an output of **mes** may either be an actual string or a symbol \perp , meaning that no message is sent to a party/functionality in this round. We write $m_{i,j}$ whenever we mean that a message was sent from party \mathcal{P}_i to \mathcal{P}_j . Similarly, we write $mres_{i,q}$ when the message was sent from \mathcal{P}_i to \mathcal{F}_q and $mres_{q,i}$ when sent from \mathcal{F}_q to \mathcal{P}_i . We 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 $\mathcal{M}_{i, \cdot}$. We 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.

Each \mathcal{P}_i has input $x_i \in \mathcal{X}$ as well as common public input $s \in \{0, 1\}^{\text{poly}(\kappa)}$.

Input $_i$: Party \mathcal{P}_i samples $r_i \xleftarrow{\$} \{0, 1\}^{\text{poly}(\kappa)}$ 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

$$\left(\{m_{i,j}^{(\tau, \rho)}\}_{j \in [n]_i}, \{mres_{i,q}^{(\tau, \rho)}\}_{q \in [r]} \right) \leftarrow \mathbf{mes}(i, x_i, r_i, s, \tau, \rho, \mathcal{M}_{\cdot, i}, \mathcal{N}_{\cdot, i}).$$

2. For each $j \in [n]_i$ send $m_{i,j}^{(\tau, \rho)}$ to \mathcal{P}_j . For each $q \in [r]$ send $mres_{i,q}^{(\tau, \rho)}$ to \mathcal{F}_q .
3. For each $j \in [n]_i$ get $m_{j,i}^{(\tau, \rho)}$ from each \mathcal{P}_j and $mres_{q,i}^{(\tau, \rho)}$ from each \mathcal{F}_q for $q \in [r]$.
4. Set $\mathcal{M}_{\cdot, i} \leftarrow \mathcal{M}_{\cdot, i} \cup \{m_{j,i}^{(\tau, \rho)}\}_{j \in [n]_i}$ and $\mathcal{N}_{\cdot, i} \leftarrow \mathcal{N}_{\cdot, i} \cup \{mres_{q,i}^{(\tau, \rho)}\}_{q \in [r]}$.

Output $_i^{(\tau)}$: Party \mathcal{P}_i computes and outputs $y_i^{(\tau)} \leftarrow \mathbf{out}(i, x_i, r_i, s, \tau, \mathcal{M}_{\cdot, i}, \mathcal{N}_{\cdot, i})$.

Fig. 1. The generic protocol II.

Communication and Output Model: We do not restrict how messages are exchanged, except that their length is polynomial in κ . If messages are sent through point-to-point secure channels, then we call this *private communication*. If 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 do not restrict the output $y_i^{(\tau)}$ which each party obtains in the end of the computation, meaning that all the $y_i^{(\tau)}$ might be different.

Universal Composition of Secure Protocols. In this work we use the (Global) Universal Composability or (G)UC model [14, 16] for analyzing security. We focus on dishonest-majority protocols as e.g. honest-majority protocols can have all parties vote on the result (if broadcast is available). Protocols are run by interactive Turing Machines (iTMs) which we call *parties*. We assume that each party \mathcal{P}_i in Π runs in PPT in the implicit security parameter κ . The PPT adversary \mathcal{A} will be able to corrupt k out of the n parties, denoted as $I \subset \mathcal{P}$. 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. Parties can exchange messages with each other and also with PPT resources, also called *ideal functionalities*. To simplify notation we assume that the messages between parties are sent over secure channels.

We start with protocols that are already UC-secure, but not verifiable. For this, we assume that the ideal functionality \mathcal{F} of a protocol Π follows the pattern described in Fig. 2: following Fig. 1 we consider protocols with one input and G output rounds. This is general enough to e.g. model commitment schemes. At the same time, our setting is not strong enough to permit reactive computations which inherently make the notation a lot more complex.

Functionality \mathcal{F} has common public input $s \in \{0, 1\}^{\text{poly}(\kappa)}$ 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 k parties where $k < n$. 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 (INPUT, sid, x_i) by \mathcal{P}_i and (INPUT, sid) by all other parties store $x_i \in \mathcal{X}$ locally and send (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^(τ): On input (COMPUTE, sid, τ) 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 $y_1^{(\tau)}, \dots, y_n^{(\tau)}$. Only this interface is allowed to alter these.
2. Send (COMPUTE, sid, τ) to every party in J_τ .

Every further call to **Compute^(τ)** 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} ^(τ): On input (OUTPUT, sid, τ) by \mathcal{P}_i where $\tau \in [G]$ and if $y_i^{(\tau)}$ was set send (OUTPUT, $sid, \tau, y_i^{(\tau)}$) to \mathcal{P}_i .

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

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} ^(τ)** are fixed in their semantics, the application may freely vary how **Compute^(τ)** may act upon the inputs or generate outputs.

The only constraint is that for each of the $\tau \in [G]$ rounds, **Compute**^(τ) sets output values $(y_1^{(\tau)}, \dots, y_n^{(\tau)})$.

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 gain full control over I . To define security, let $\Pi \circ \mathcal{A}$ be the distribution of the output of \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 Π securely implements \mathcal{F} if for every PPT iTM \mathcal{A} that maliciously corrupts at most k parties 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 κ .*

In our (public and secret) Join Authentication protocols we use the standard functionalities for digital signatures \mathcal{F}_{Sig} [15] and for key registration functionality \mathcal{F}_{Reg} [17]. Moreover, to simplify our compiler description, we use an authenticated bulletin board functionality \mathcal{F}_{BB} described in the full version [6].

2.1 Verifiable Functionalities

We extend the functionality \mathcal{F} from Sect. 2 to provide a notion of non-interactive verification using a functionality wrapper \mathcal{F}^{V} described in Fig. 3. For this, we assume that there are additional parties \mathcal{V}_i which can partake in the verification. These, as well as regular protocol parties, can register at runtime to be verifiers of the computation using a special interface **Register Verifier**. Once they are registered, these verifiers are allowed to check the validity of outputs for parties that have initiated verification at any point. We keep track of this using the set of verifiers \mathcal{V} (which is initially empty) inside the functionality. For values whose output has been provided using the interface **Output** _{i} ^(τ) (that we inherit from the definition of \mathcal{F} of Sect. 2) we allow the parties \mathcal{P} to use an interface called **Activate Verification** to enable everyone in \mathcal{V} to check their outputs via the interface **Verify** _{i} . The modifications to **Input** _{i} and the new interface **NMF** _{\mathcal{S}^{V}} are related to the über simulators discussed in Appendix 3.2.

Notice that, in our constructions, a verifier $\mathcal{V}_i \in \mathcal{V}$ can perform verification with help from data obtained in 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. reading verification data from publicly available resource such as \mathcal{F}_{BB} . In case \mathcal{V}_i obtains 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). When \mathcal{V}_i obtains verification data from a setup resource that is untamperable by the adversary, \mathcal{S} has no influence on the verification process. To model these cases, \mathcal{F}^{V} might implement only **Register Verifier (public)** or only **Register Verifier (private)**, respectively. We do not require \mathcal{F}^{V} to

The functionality wrapper $\mathcal{F}^v[\mathcal{F}]$ adds the interfaces below to a generic functionality \mathcal{F} defined as in Figure 2, still allowing direct access to \mathcal{F} . \mathcal{F}^v is parameterized by an über simulator \mathcal{S}^u executed internally (as discussed in Appendix 3.2) and maintains binary variables **verification-active**, **verify-1**, \dots , **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}^v interacts with verifiers $\mathcal{V}_i \in \mathcal{V}$.

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

- If **verification-active** = 1 send (REGISTER, sid, \mathcal{V}_i) to \mathcal{S} . If \mathcal{S} answers with (REGISTER, sid, \mathcal{V}_i, ok), set $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_i$ and return (REGISTERED, sid) to \mathcal{V}_i .
- If **verification-active** = 0 return (VERIFICATION-INACTIVE, sid) to \mathcal{V}_i .

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

- If **verification-active** = 1 set $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_i$ and return (REGISTERED, sid) to \mathcal{V}_i .
- If **verification-active** = 0 return (VERIFICATION-INACTIVE, sid) to \mathcal{V}_i .

Activate Verification: Upon receiving (ACTIVATE-VERIFICATION, sid , **open-i**, **open-input-i**) from each \mathcal{P}_i and if **Compute**⁽¹⁾, \dots , **Compute**^(G) succeeded:

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

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

- if **verify-j** = 1 then compute the set $B \leftarrow \{\tau \in [G] \mid b^{(\tau)} \neq \mathbf{y}_j^{(\tau)}\}$. If $a = z_j$, then set $f \leftarrow$ 1; otherwise $f \leftarrow$ 0. Return (VERIFY, sid, j, f, B) to \mathcal{V}_i .
- If **verify-j** = 0 then send (CANNOT-VERIFY, sid, j) to \mathcal{V}_i .

Input_i: On input (INPUT, sid, x_i) by \mathcal{P}_i and (INPUT, sid) by all other parties, forward (INPUT, sid, x_i) to \mathcal{F} and also forward responses from \mathcal{F} to \mathcal{P}_i . Finally, after receiving (INPUT, sid, x_i) from all $\mathcal{P}_i, i \in \bar{I}$ (i.e. all honest parties), initialize \mathcal{S}^u parameterizing it with \mathcal{F} 's randomness tape and with x_i for all honest \mathcal{P}_i .

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

Fig. 3. The Functionality wrapper $\mathcal{F}^v[\mathcal{F}]$. The modifications to interface **Input_i** and the new interface **NMF_{S^u}** are discussed in Appendix 3.2.

implement both of these interfaces simultaneously, and thus define the properties of \mathcal{F}^V according to which of them is present:

- A functionality which implements the interface **Register Verifier (public)** is said to have *Public Verifier Registration*. We say that it has *Private Verifier Registration* if it implements **Register Verifier (private)**
- A functionality which implements the interfaces **Activate Verification** and **Verify_j** and which has *Verifier Registration* is called Non-Interactively Verifiable (NIV). If it has *Public Verifier Registration* then it is *Publicly Verifiable*, if it has *Private Verifier Registration* it is *Privately Verifiable*

3 Verifiable Protocols

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

Our approach to verification (as outlined in Sect. 1.3) is to leverage properties for verifiability that are already built into a protocol Π . As the verifier can only rely on the protocol transcript, consider how such a transcript comes into existence: we first run an instance of Π with an adversary \mathcal{A} . Afterwards, the adversary may change parts of the protocol transcript in order to trigger faulty behavior in the outputs of parties. If the adversary cannot trigger erroneous behavior this way, 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 $\Pi = (\mathbf{nmes}, \mathbf{out})$.

Transcript Validity: If our verification relies on the transcript of Π , then a transcript is incorrect if messages that a party \mathcal{P}_i claims to have sent were not received by receiving party \mathcal{P}_j , if messages to and from a NIV functionality \mathcal{F}^V were not actually sent or received by \mathcal{P}_i or if, in case a party \mathcal{P}_i reveals 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 \mathbf{nmes} and the remaining transcript. We formalize this as *Transcript Validity* in the full version [6].

Transcript Non-Malleability: Tampering of an adversary with the transcript can be ok unless it leads to two self-consistent protocol transcripts with outputs $\hat{y}_i^{(\tau)} \neq y_i^{(\tau)}$ for some \mathcal{P}_i such that both $\hat{y}_i^{(\tau)}, y_i^{(\tau)} \neq \perp$. To prevent 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 describe a security game that constrains \mathcal{A} beyond transcript validity:

1. \mathcal{A} runs the protocol with a challenger \mathcal{C} , which simulates honest parties whose inputs and randomness \mathcal{A} does not know, generating a transcript τ .

2. The adversary will obtain some additional potentially secret information of the honest parties from \mathcal{C} , upon which it outputs two valid protocol transcripts Π_0, Π_1 .
3. \mathcal{A} wins if Π_0, Π_1 coincide in certain parts with τ , while the outputs of some party \mathcal{P}_i are different and not \perp .

We want to cover a diverse range of protocols which might come with different 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 $\nu = \text{ncir}$ 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 $I \cap \text{RIR}$ might be altered by the adversary. If instead $\nu = \text{cir}$ then all parties *are committed* to the input and randomness in the beginning of the execution and the adversary cannot alter x_i, r_i revealed for verification by honest or dishonest parties from RIR. For μ we give the adversary the following choices:

- $\mu = \text{ncmes}$: \mathcal{A} can replace all messages *by all parties*.
- $\mu = \text{chsmes}$: \mathcal{A} can replace messages *from corrupted senders*.
- $\mu = \text{chmes}$: \mathcal{A} can replace messages exchanged *between corrupted parties*.
- $\mu = \text{cmes}$: \mathcal{A} cannot replace *any message*.

The full definition of *Transcript Non-Malleability* is given in the full version [6].

3.1 Simulating Verifiable Protocols: Input-Aware Simulation

Most 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, \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 Sect. 1.1. Basically, in the publicly verifiable version of a protocol, we wish to embed a special simulator into the publicly verifiable functionality that it realizes. This allows to “delegate” the simulation of the compiled protocol, while the simulator for the publicly verifiable version handles the machinery needed to obtain public verifiability. This 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}^u : We now introduce the notion of an *über simulator* for a UC-secure protocol Π realizing a functionality \mathcal{F} . We denote über simulators as \mathcal{S}^u , while we denote by \mathcal{S} the original simulator used in the original UC proof. Basically, an über simulator \mathcal{S}^u takes the inputs to be used by simulated honest parties, as well as the randomness of the functionality, as an external parameter, and uses these in interactions with the adversary. It furthermore 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. Moreover, an über simulator allows for responses to queries to simulated functionalities to be given externally. Otherwise, \mathcal{S}^u operates like a regular simulator, *e.g.* extracting corrupted parties' inputs.

In the case of a probabilistic functionality \mathcal{F} , the über simulator \mathcal{S}^u also receives the randomness tape used by \mathcal{F} . \mathcal{S}^u uses this tape to determine the random values that will be sampled by \mathcal{F} , simulating an execution compatible with such values and with the inputs from honest parties (if they have any).

Most existing simulators for protocols realizing the vast majority of natural UC functionalities can be trivially modified to obtain an über simulator which we explain in the full version [6]. This is because they basically execute the protocol as an honest party would, except that they use random inputs and leverage the setup to equivocate the output in the simulated execution. Departing from such a simulator, an über simulator can be constructed by allowing the simulated honest party inputs to be obtained externally, rather than generated randomly.

Syntax of Über Simulator \mathcal{S}^u : Let \mathcal{S}^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 a copy of an adversary \mathcal{A} (either connected to \mathcal{A} or to \mathcal{F}^v 's $\text{NMF}_{\mathcal{S}^v}$ interface) and a tape for messages from the global ideal functionalities in the hybrid model where Π is defined. If \mathcal{F} is probabilistic, \mathcal{S}^u also receives \mathcal{F} 's random tape.
- **Output tapes:** tapes for output to \mathcal{Z} , tapes for messages to \mathcal{F} , \mathcal{A} , tapes for messages for the global 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.

For any PPT iTM \mathcal{S}^u with the input and output tapes defined above, we then say that \mathcal{S}^u is an über simulator if it has the properties of *simulation- and execution-consistency*, which are described in Definitions 2 and 3 below. Simulation consistency says that any regular ideal world execution of \mathcal{F} with \mathcal{S} is indistinguishable from an execution of \mathcal{F} with \mathcal{S}^u where \mathcal{S}^u operates as \mathcal{S} does (*i.e.* with direct access to a copy of the adversary \mathcal{A} and the global setup) but is parameterized by the dummy honest party inputs instead of choosing simulated honest party inputs at random. Formally, simulation consistency is as follows:

Definition 2 (Simulation Consistency). *Let Π be a protocol UC-realizing functionality \mathcal{F} using ideal functionalities $\mathcal{F}_1, \dots, \mathcal{F}_r$ as setup and let \mathcal{S} be the*

simulator of \mathcal{F} 's proof. We say that the PPT iTM \mathcal{S}^U is Simulation-consistent for $(\Pi, \mathcal{F}, \mathcal{S})$ if these distributions are indistinguishable for all PPT iTM \mathcal{Z} :

1. $\mathcal{F} \circ \mathcal{S}$: The distribution of outputs of \mathcal{Z} in an ideal execution of \mathcal{F} and \mathcal{S} executing an internal copy of adversary \mathcal{A} and potentially a set of global functionalities.
2. $\mathcal{F} \circ \mathcal{S}^U$: The distribution of outputs of \mathcal{Z} in an ideal execution of \mathcal{F} with \mathcal{S}^U , where \mathcal{S}^U 's corresponding input/output tapes are connected directly to a copy of \mathcal{A} and to global setup functionalities (instead of \mathcal{F}^V 's $\text{NMF}_{\mathcal{S}^V}$ interface). \mathcal{S}^U 's tapes for simulated honest party inputs are initialized with the same inputs that are provided by the dummy honest parties to \mathcal{F} and \mathcal{S}^U is given a uniformly random tape.

\mathcal{Z} gives inputs to all parties as in the standard UC simulation experiment but only has access to the same input/output tapes of \mathcal{S}^U that it can access for \mathcal{S} .

We remark that \mathcal{S}^U does not have two explicitly different modes of operations depending on whether it is executed inside \mathcal{F}^V or in the experiment of Definition 2. In both scenarios, \mathcal{S}^U has the same input/output tapes and access to \mathcal{F} 's interfaces, with the sole differences being its input/output tapes for a copy of the adversary being either directly connected to the adversary in the experiment of Definition 2 or to \mathcal{F}^V 's $\text{NMF}_{\mathcal{S}^V}$ interface and its input/output tapes for global setup functionalities being connected to these functionalities in the experiment of Definition 2 or to \mathcal{F}^V 's $\text{NMF}_{\mathcal{S}^V}$ interface. This observation is important when arguing why \mathcal{S}^U does not give \mathcal{F}^V 's ideal adversary (*i.e.* \mathcal{F}^V 's simulator) any undue advantage by, *e.g.*, leaking information about honest parties' inputs.

Execution consistency states that the randomness for simulated honest parties output by an über simulator \mathcal{S}^U parameterized with the same inputs as the real honest parties must be consistent with the randomness of a real protocol execution. We use the following formal definition:

Definition 3 (Execution Consistency). *Let Π be a UC-secure implementation of the functionality \mathcal{F} in the $\mathcal{F}_1, \dots, \mathcal{F}_r$ -hybrid model and let \mathcal{S} be the simulator of the proof. We say that the PPT iTM \mathcal{S}^U is Execution-consistent for $(\Pi, \mathcal{F}, \mathcal{S})$ if for all PPT iTM \mathcal{Z} and PPT iTM \mathcal{A} the following distributions are indistinguishable:*

1. The distribution of outputs of \mathcal{Z} in a real execution of Π with adversary \mathcal{A} and honest parties $\mathcal{P}_1, \dots, \mathcal{P}_k$ whose input and randomness pairs are $(x_{h_1}, R_{h_1}), \dots, (x_{h_k}, R_{h_k})$ in the $\mathcal{F}_1, \dots, \mathcal{F}_r$ -hybrid model. The tuple of honest party randomness $(R_{h_1}, \dots, R_{h_k})$ is output by \mathcal{S}^U after an execution with \mathcal{F} where \mathcal{S}^U interacts with a copy of \mathcal{A} and \mathcal{S}^U 's tapes for simulated honest party inputs are initialized with the same honest party inputs $(x_{h_1}, \dots, x_{h_k})$ as those given to $\mathcal{P}_1, \dots, \mathcal{P}_k$.
2. The distribution of outputs of \mathcal{Z} in a real execution of Π with adversary \mathcal{A} and honest parties $\mathcal{P}_1, \dots, \mathcal{P}_k$ with inputs $(x_{h_1}, \dots, x_{h_k})$ in the $\mathcal{F}_1, \dots, \mathcal{F}_r$ -hybrid model where honest party randomness is sampled by \mathcal{Z} .

\mathcal{Z} gives inputs to all parties in both the ideal and real executions as in the standard UC simulation experiment, the difference being that in 1. honest party randomness is provided by \mathcal{S}^U and in 2. it is sampled by \mathcal{Z} .

For any PPT iTM \mathcal{S}^U with the input and output tapes defined above we then say that \mathcal{S}^U is an über simulator if it is simulation- and execution-consistent. We summarize this in the full version [6].

Über Simulators with Global Setup: In order to argue that \mathcal{S}^U does not leak any information on honest parties' inputs through \mathcal{F}^V 's $\text{NMF}_{\mathcal{S}^U}$ interface, we will restrict the class of global functionalities that can be used as setup in compiled protocols. Intuitively, we require that all global functionalities used by a protocol with a global simulator provide all parties with access to the same interface and answers queries from all parties with the same answer (e.g. in a global random oracle). This is necessary both for practical and technical reasons: 1-(practical) the verification procedure of our compiler needs the same access to global setup as the party who activated verification and revealed its input/randomness; 2-(technical) \mathcal{S}^U must not be able to distinguish whether it is operating within \mathcal{F}^V or within the experiment of Definition 2. In order to achieve these goals, we introduce the notion of Admissible Global Setup in Definition 4 and restrict our compiler to work only on protocols with Admissible Global Setup.

Definition 4 (Admissible Global Setup). *A global ideal functionality \mathcal{G} is admissible if:*

- All parties $\mathcal{P}_i \in \mathcal{P}$ have access to the same interfaces (i.e. all parties can send the same queries to \mathcal{G}).
- For all of \mathcal{G} 's interfaces, for all possible queries Q , there exists a single response R such that, upon receiving a query Q_j from any party $\mathcal{P}_i \in \mathcal{P}$, \mathcal{G} returns R .

3.2 Functionalities \mathcal{F}^V with Embedded Über Simulator \mathcal{S}^U

We now outline how an über simulator \mathcal{S}^U (Definition in the full version [6]) for the protocol Π will be used with a functionality \mathcal{F}^V . Note that \mathcal{S}^U is internally executed by the functionality wrapper \mathcal{F}^V presented in Fig. 3, which can be accessed by an ideal adversary (i.e. \mathcal{F}^V 's Simulator) interacting with \mathcal{F}^V through interfaces **Input** _{i} and $\text{NMF}_{\mathcal{S}^U}$. Moreover, \mathcal{F}^V allows \mathcal{S}^U to query admissible global setup functionalities $\mathcal{F}_1, \dots, \mathcal{F}_n$ (according to Definition 4) on behalf of honest parties.

The internal \mathcal{S}^U executed by \mathcal{F}^V takes care of simulating the original protocol Π that realizes \mathcal{F} being compiled into a publicly verifiable protocol Π^V that realizes $\mathcal{F}^V[\mathcal{F}]$, while the external \mathcal{S}^V interacting with \mathcal{F}^V will take care of simulating the additional protocol steps and building blocks used in obtaining public verifiability in Π^V . In order to do so, \mathcal{F}^V will parameterize \mathcal{S}^U with the inputs of all honest parties \mathcal{P}_i , which are received through interface **Input** _{i} , as well as the randomness of \mathcal{F} if the functionality is probabilistic. As the execution

progresses, \mathcal{S}^V executes the compiled protocol Π^V with an internal copy \mathcal{A} of the adversary and extracts the messages of the original protocol Π from this execution, forwarding these messages to \mathcal{S}^U through the interface $\mathbf{NMF}_{\mathcal{S}^V}$. Moreover, \mathcal{S}^V will provide answers to queries to setup functionalities from \mathcal{A} as instructed by \mathcal{S}^U also through interface $\mathbf{NMF}_{\mathcal{S}^V}$. All the while, queries from honest parties simulated by \mathcal{S}^U to setup functionalities are directly forwarded back and forth by \mathcal{F}^V . If verification is ever activated by an honest party \mathcal{P}_i (and $\mathcal{P}_i \in \text{RIR}$), \mathcal{F}^V not only leaks that party's input to \mathcal{S}^V but also leaks that party's randomness R_{h_i} in the simulated execution with \mathcal{S}^U (provided by \mathcal{S}^U). As we discuss in Sect. 5, this will allow \mathcal{S}^V to simulate verification, since it now has both a valid transcript of an execution of Π^V with \mathcal{A} and a matching input and randomness pair that matches that transcript (provided by \mathcal{F}^V with the help of \mathcal{S}^U).

We remark that this strategy does not give the simulator \mathcal{S}^V any extra power in simulating an execution of the compiled protocol Π^V towards \mathcal{A} other than the power the simulator \mathcal{S} for the original protocol Π already has. We will establish that the access to \mathcal{S}^U given by \mathcal{F}^V to \mathcal{S}^V does not allow it to obtain any information about the inputs of honest parties. Notice that in an execution with admissible global setup (according to Definition 4), the only difference between \mathcal{S}^U 's execution within \mathcal{F}^V and \mathcal{S}^U 's execution in the experiment of Definition 2 is that, when it is executed within \mathcal{F}^V , its input/output tapes for a copy of the adversary are connected to \mathcal{S}^U via the $\mathbf{NMF}_{\mathcal{S}^V}$ interface. Hence, the only way \mathcal{S}^U can detect that it is being executed within \mathcal{F}^V and leak any undue information is via its interaction via the adversary input/output tapes. However, the definition in the full version [6] establishes that this interaction is indistinguishable from that of the original simulator \mathcal{S} for protocol Π . Since Π is UC-secure, an execution of \mathcal{F} with \mathcal{S} does not leak any information about the simulated parties' inputs (*i.e.* inputs randomly picked by \mathcal{S}), which would trivially allow \mathcal{Z} to distinguish an execution of \mathcal{F} with \mathcal{S} from a real world execution of Π with \mathcal{A} . Thus, by the definition of an über simulator in the full version [6] and the UC security of Π , \mathcal{S}^U does not leak any information about honest party inputs to \mathcal{S}^V via interface $\mathbf{NMF}_{\mathcal{S}^V}$ when executed within \mathcal{F}^V .

4 Joint Authentication Functionalities

We now define authentication functionalities that serve as building blocks for our compiler. These 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 extend this functionality to allow for joint authentication of *private* messages that are only known in encrypted form.

As opposed to classical point-to-point or broadcast authenticated channels, our functionalities do not deliver messages to the 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 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, i.e. our notion of *transcript non-malleability*.

Public Joint Authentication: First, consider the simple case of authenticating public messages (known by all parties participating in the joint authentication procedure). Here, 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 to the functionality. This can be implemented 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 (this is captured in the functionality). Functionality $\mathcal{F}_{\text{PJAuth}}$ is described in the full version [6].

Secret Joint Authentication: We further define a functionality $\mathcal{F}_{\text{SJAuth}}$ (described in the full version [6]). This functionality works similarly to $\mathcal{F}_{\text{PJAuth}}$, 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 these plaintext messages. If a private message is revealed by its sender (or one of its receivers), $\mathcal{F}_{\text{SJAuth}}$ allows third parties (including the bureaucrats) to verify that this message is indeed the one that was authenticated.

In order to capture the different actions of each party it interacts with, $\mathcal{F}_{\text{SJAuth}}$ is parameterized by the following (sets of) parties: a sender \mathcal{P}_{snd} that can input messages to be jointly authenticated; a set of parties \mathcal{P} who receive input messages from \mathcal{P}_{snd} and jointly authenticate them; a set of bureaucrats \mathcal{B} who jointly authenticate that \mathcal{P}_{snd} has sent a certain (unknown to them) committed message to \mathcal{P} . $\mathcal{F}_{\text{SJAuth}}$, like $\mathcal{F}_{\text{PJAuth}}$, does not aid in sending messages, notifications about sent messages nor joint authentication information to any party. The responsibility for sending messages lies with \mathcal{P}_{snd} , while \mathcal{P}_{snd} or $\mathcal{P}_i \in \mathcal{P}$ can notify other parties that plaintext verification is possible.

We realize $\mathcal{F}_{\text{SJAuth}}$ with a signature scheme and a certified encryption scheme with plaintext verification, i.e. an encryption scheme with two properties: (1) all parties' public keys are registered in a PKI, making sure that encrypted messages can only be opened by the intended receiver; (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. To prove that a certain message was indeed contained in the ciphertext, the receiver(s) recovers the plaintext message and a

proof of plaintext validity from the ciphertext and sends those to the verifier(s). Finally, a verifier first checks that the message was indeed contained in a previously sent ciphertext and that this ciphertext has been signed by all parties in \mathcal{P} and \mathcal{B} . This construction and a concrete realization are described in the full version [6].

Authenticating Inputs and Randomness: To provide an authentication of inputs and randomness we adapt the functionality $\mathcal{F}_{\text{SJAuth}}$, as the desired capabilities are like a message authentication without a receiver. In the full version [6] we present a functionality $\mathcal{F}_{\text{IRAuth}}$ that implements this.

5 Compilation for Input-Revealing Protocols

We now sketch how to compile protocols from Sect. 2 into non-interactively verifiable counterparts. As we focus on protocols according to the definition in the full version [6] there are 8 combinations of parameters (ν, μ) for (ν, RIR, μ) -transcript non-malleable protocols to consider. Furthermore we might either have public or private verifier registration, which in total yields 16 different definitions. To avoid redundancy we now outline how to get the respective verifiability in each setting. We simplify notation by just using a single verifier \mathcal{V} .

Assume a (ν, RIR, μ) -transcript non-malleable protocol Π that UC realizes the functionality \mathcal{F} in the (global) $\mathcal{F}_1, \dots, \mathcal{F}_r$ -hybrid model with über simulator \mathcal{S}^U for $(\Pi, \mathcal{F}, \mathcal{S})$. Then compilation works as follows:

1. We describe how to construct a protocol Π^V by modifying Π with access to a signature functionality \mathcal{F}_{Sig} , a key registration functionality \mathcal{F}_{Reg} and authentication functionalities $\mathcal{F}_{\text{PJAuth}}, \mathcal{F}_{\text{SJAuth}}, \mathcal{F}_{\text{IRAuth}}$. We will furthermore require that we can replace the hybrid functionalities $\mathcal{F}_1, \dots, \mathcal{F}_r$ used in Π with verifiable counterparts $\mathcal{F}_1^V, \dots, \mathcal{F}_r^V$.
2. In Appendix 3.2 we show how Π^V UC-realizes $\mathcal{F}^V[\mathcal{F}]$ in the $\mathcal{F}_1^V, \dots, \mathcal{F}_r^V$ -hybrid by constructing a simulator \mathcal{S}^V .

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 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 :

$\mu = \text{ncmes}$: The adversary can replace all messages by any party at his will, and messages are just exchanged as in Π .

$\mu = \text{chsmes}$: Before the protocol begins, each \mathcal{P}_i generates a signing key with \mathcal{F}_{Sig} and registers its signing key with \mathcal{F}_{Reg} . Whenever \mathcal{P}_i sends a message $\mathbf{m}_{i,j}$ to \mathcal{P}_j it uses \mathcal{F}_{Sig} to authenticate $\mathbf{m}_{i,j}$ with a signature $\sigma_{i,j}$. \mathcal{V} will later be able to verify exactly those messages that were sent by honest parties, as \mathcal{A} can fake messages and signatures sent by dishonest parties.

$\mu = \text{chmes}$: Each message that is either sent or received by an honest party must remain unaltered. Each party will do the same as for $\mu = \text{chsmes}$, but whenever \mathcal{P}_i receives a message $\mathbf{m}_{j,i}$ from \mathcal{P}_j then it uses \mathcal{F}_{Sig} to authenticate $\mathbf{m}_{j,i}$ with a signature $\sigma_{j,i}$. Now \mathcal{V} can establish for each message of the protocol if both sender and receiver signed the same message. \mathcal{A} can only alter messages that were both sent and received by dishonest parties.

$\mu = \text{cmes}$: Here dishonest parties cannot replace their messages before verification. To achieve this, we use $\mathcal{F}_{\text{SJAuth}}, \mathcal{F}_{\text{PJAuth}}$ as defined in Sect. 4 which the parties now use to register their private message exchange. These functionalities can then be used by \mathcal{V} to validate the transcript.

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

$\nu = \text{cir}$: The parties use $\mathcal{F}_{\text{IRAuth}}$ to authenticate their inputs and randomness. \mathcal{V} uses $\mathcal{F}_{\text{IRAuth}}$ to check validity of the revealed x_i, r_i which it obtained.

Hybrid Functionalities: Replace the hybrid functionalities $\mathcal{F}_1, \dots, \mathcal{F}_r$ with NIV counterparts, i.e. with functionalities $\mathcal{F}_1^v, \dots, \mathcal{F}_r^v$ that have the same interfaces as defined in Sect. 2.1. To achieve public verifiability each such \mathcal{F}_q^v must also be publicly verifiable. If a global functionality is used as setup, it must be admissible according to Definition 4, so that the verification procedure can re-execute the protocol. For any such \mathcal{F}_q^v , \mathcal{V} can establish if a message $\mathbf{mres}_{q,i}$ was indeed sent to \mathcal{P}_i or not. If \mathcal{F}_q^v does also reveal inputs, it can also test if $\mathbf{mres}_{i,q}$ as claimed to be sent by \mathcal{P}_i was indeed received by the functionality.

Public Verifiability Compiler. The basic idea is to turn any (ν, RIR, μ) -transcript non-malleable protocol into a $(\text{cir}, \text{RIR}, \mu)$ -transcript non-malleable protocol by forcing the adversary to commit to all the corrupted parties' randomness and inputs, and then turn it into a $(\text{cir}, \text{RIR}, \text{cmes})$ -transcript non-malleable protocol by forcing the adversary to commit to all messages. While this might be overkill for some protocols, we focus on the worst case scenario of compiling

(ncir, RIR, ncmes)-transcript non-malleable protocols, since it is the most challenging. After making a protocol (cir, RIR, cmes)-transcript non-malleable, the protocol execution becomes deterministic and can be verified upon 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. We present a detailed description of this compiler and a formal theorem statement together with its proof in the full version [6].

References

1. Alwen, J., Ostrovsky, R., Zhou, H.-S., Zikas, V.: Incoercible multi-party computation and universally composable receipt-free voting. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 763–780. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_37
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multi-party computations on bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 443–458. IEEE Computer Society Press, May 2014
3. Asharov, G., Orlandi, C.: Calling out cheaters: covert security with public verifiability. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 681–698. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34961-4_41
4. Baum, C., Damgård, I., Orlandi, C.: Publicly auditable secure multi-party computation. In: Abdalla, M., De Prisco, R. (eds.) SCN 2014. LNCS, vol. 8642, pp. 175–196. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10879-7_11
5. Baum, C., David, B., Dowsley, R.: Insured MPC: efficient secure computation with financial penalties. In: Bonneau, J., Heninger, N. (eds.) FC 2020. LNCS, vol. 12059, pp. 404–420. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51280-4_22
6. Baum, C., David, B., Dowsley, R.: (Public) verifiability for composable protocols without adaptivity or zero-knowledge. Cryptology ePrint Archive, Paper 2020/207 (2020). <https://eprint.iacr.org/2020/207>
7. Baum, C., David, B., Frederiksen, T.K.: P2DEX: privacy-preserving decentralized cryptocurrency exchange. In: Sako, K., Tapp, N.O. (eds.) ACNS 2021. LNCS, vol. 12726, pp. 163–194. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78372-3_7
8. Baum, C., Orsini, E., Scholl, P.: Efficient secure multiparty computation with identifiable abort. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9985, pp. 461–490. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53641-4_18
9. Baum, C., Orsini, E., Scholl, P., Soria-Vazquez, E.: Efficient constant-round MPC with identifiable abort and public verifiability. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12171, pp. 562–592. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56880-1_20
10. Bellare, M., Hoang, V.T., Rogaway, P.: Adaptively secure garbling with applications to one-time programs and secure outsourcing. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 134–153. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34961-4_10

11. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_24
12. Camenisch, J., Dubovitskaya, M., Rial, A.: UC commitments for modular protocol design and applications to revocation and attribute tokens. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9816, pp. 208–239. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_8
13. Camenisch, J., Lehmann, A., Neven, G., Samelin, K.: UC-secure non-interactive public-key encryption. In: IEEE CSF 2017 (2017)
14. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: 42nd FOCS, pp. 136–145. IEEE Computer Society Press, October 2001
15. Canetti, R.: Universally composable signature, certification, and authentication. In: CSFW 2004 (2004)
16. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_4
17. Canetti, R., Herzog, J.: Universally composable symbolic analysis of mutual authentication and key-exchange protocols. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 380–403. Springer, Heidelberg (2006). https://doi.org/10.1007/11681878_20
18. Canetti, R., Jain, A., Scafuro, A.: Practical UC security with a global random oracle. In: Ahn, G.-J., Yung, M., Li, N. (eds.) ACM CCS 2014, pp. 597–608. ACM Press, November 2014
19. Canetti, R., Rabin, T.: Universal composition with joint state. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 265–281. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_16
20. Canetti, R., Sarkar, P., Wang, X.: Blazing fast OT for three-round UC OT extension. In: Kiayias, A., Kohlweiss, M., Wallden, P., Zikas, V. (eds.) PKC 2020. LNCS, vol. 12111, pp. 299–327. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45388-6_11
21. Canetti, R., Sarkar, P., Wang, X.: Efficient and round-optimal oblivious transfer and commitment with adaptive security. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020. LNCS, vol. 12493, pp. 277–308. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64840-4_10
22. Cascudo, I., David, B.: SCRAPE: scalable randomness attested by public entities. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) ACNS 2017. LNCS, vol. 10355, pp. 537–556. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61204-1_27
23. Cascudo, I., David, B.: ALBATROSS: publicly Attestable BATCHed randomness based on secret sharing. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020. LNCS, vol. 12493, pp. 311–341. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64840-4_11
24. Chen, J., Micali, S.: Algorand: a secure and efficient distributed ledger. *Theor. Comput. Sci.* **777**, 155–183 (2019)
25. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: 18th ACM STOC, pp. 364–369. ACM Press, May 1986
26. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_38

27. David, B., Gaži, P., Kiayias, A., Russell, A.: Ouroboros praos: an adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10821, pp. 66–98. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_3
28. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_10
29. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC, pp. 218–229. ACM Press, May 1987
30. Ishai, Y., Ostrovsky, R., Zikas, V.: Secure multi-party computation with identifiable abort. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 369–386. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_21
31. Jafargholi, Z., Oechsner, S.: Adaptive security of practical garbling schemes. In: Bhargavan, K., Oswald, E., Prabhakaran, M. (eds.) INDOCRYPT 2020. LNCS, vol. 12578, pp. 741–762. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65277-7_33
32. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: a provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 357–388. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_12
33. Kiayias, A., Zhou, H.-S., Zikas, V.: Fair and robust multi-party computation using a global transaction ledger. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 705–734. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_25
34. Lindell, Y., Pinkas, B.: Secure two-party computation via cut-and-choose oblivious transfer. *J. Cryptol.* **25**(4), 680–722 (2012)
35. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
36. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 681–700. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_40
37. Schoenmakers, B., Veeningen, M.: Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In: Malkin, T., Kolesnikov, V., Lewko, A.B., Polychronakis, M. (eds.) ACNS 2015. LNCS, vol. 9092, pp. 3–22. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28166-7_1
38. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole - reducing data transfer in garbled circuits using half gates. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 220–250. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_8