More from Less: Composable General Multi-Party Computation with Global Public Verifiability from a Single Enclave Only*

Saskia Bayreuther, Robin Berger, Felix Dörre, Eva Hetzel, Yufan Jiang, Christian Martin, Jeremias Mechler, and Jörn Müller-Quade

KASTEL Security Research Labs, Karlsruhe Institute of Technology, Karlsruhe, Germany
{firstname.lastname}@kit.edu

Abstract. Trusted Execution Environments (TEEs), also known as secure enclaves, such as Intel SGX, Intel TDX, and AMD SEV, are seeing widespread use to perform computations on highly sensitive data. Several formal models exist to analyze the security of cryptographic protocols using TEEs, notably the one by Pass *et al.* (EUROCRYPT 2017) for *attested computations* in the Generalized UC framework. Using this model and the proposed global ideal functionality $\mathcal{G}_{\text{att}}^{\text{PST}}$, provably secure multi-party computations with practical efficiency are possible.

Attested computations are achieved by having enclave outputs signed with a key pair held by $\mathcal{G}_{\text{att}}^{\text{PST}}$, together with the enclave's code. Being a global functionality, the verification key can be obtained by *any* party. Perhaps surprisingly, this model does not give rise to a meaningful notion of public verifiability, *i.e.* the ability of external parties to plausibly verify results, even though some commercially available enclaves allow exactly that. We formalize this intuition in the form of an impossibility result and propose a novel simulation technique where equivocation is not handled by the simulator resp. adversary anymore, but in a coordinated effort between our new functionality $\mathcal{G}_{\text{att}}$ and a (local) ideal functionality $\mathcal{F}$ that is realized with public verifiability. To this end, several technical problems need to be solved, in particular to ensure that this new mechanism cannot be abused. While this approach is unconventional, to the best of our knowledge it is the first to achieve a general variant of public verifiability, a) even when *all* protocol parties are corrupted and $\mathcal{F}$ is probabilistic and, b) where guarantees of honest (external) verifiers are not affected by simulation at all. We call the latter property *global* public verifiability.

We also address a second impossibility result of Pass *et al.*, namely the requirement that every protocol party needs a TEE (even in a setting without public verifiability), unless an additional (global) setup is used.

* This article is based on an earlier article: Saskia Bayreuther et al. "More from Less: Composable General Multi-party Computation with Global Public Verifiability from a Single Enclave Only". In: *TCC 2026: 24th Theory of Cryptography Conference, Part IV*. Ed. by Dana Dachman-Soled and Daniel Wichs. Vol. 17194. Lecture Notes in Computer Science. New York City, NY, USA: Springer, Cham, Switzerland, 2027, pp. 328–361. DOI: 10.1007/978-3-032-43630-6_11, © IACR 2027.

We address this impossibility result by introducing *designated-verifier attestations* that are only valid for a single party in a single protocol execution, akin to what is possible with real-world enclaves.
Using our improved model, we propose protocols for (globally publicly verifiable) composable general MPC and prove their security under the notion of Universal Composition with Global Subroutines (Badertscher *et al.*, TCC 2020) and static malicious corruptions.

Keywords: Secure Enclaves · Public Verifiability · Universal Composability.

1 Introduction

Ever since the seminal results of Goldreich, Micali, and Wigderson [GMW87] and Yao [Yao86], protocols to securely evaluate (almost) every efficient functionality are known. However, despite 40 years of research, techniques for secure multi-party computation (MPC) are still several orders of magnitude too inefficient to handle big data or AI use-cases.

To still perform such tasks with technical security guarantees, hardware-based solutions like *trusted execution environments* (TEEs), also known as *secure enclaves*, have been proposed. Enclaves, as part of a system's CPU or GPU, aim to provide a secure execution environment in an otherwise untrusted system, including the ability to execute arbitrary (almost) unmodified code. Often, such enclaves are used as trusted third parties for outsourcing: Users send their encrypted inputs, the enclave performs a computation on the plaintexts and returns the encrypted results. As the used cryptography is lightweight and the computations are performed natively, the performance of this approach is typically very high. To establish a key with an enclave and to receive evidence that the expected code is running, many enclaves feature so-called *remote attestations*.

Interestingly, this remote attestation may not only be meaningful for parties that participate in the computation, but also for parties that have not participated and merely want to verify the (presumptive) results. Examples include electronic voting where election results should be verifiable by third parties or certain outsourced computations like the training of a machine-learning model, the choice of sufficiently unbiased randomness for the use in scientific experiments, or even (non-interactive) zero-knowledge proofs of knowledge.

Given the power of remote attestation of some real-world enclaves like Intel SGX or Intel TDX, the above use-cases can, in principle, be solved with enclaves (if their real-world security is deemed to be adequate for the respective use-case).

Real-World Enclaves. Real-world enclaves like Intel SGX [CD16], Intel TDX [Che+24] or AMD SEV-SNP [Mis+25], allow their users to run arbitrary programs, provide input to them, and then perform remote attestation to convince a third party that a specific result is the output of a specific program. Remote attestation is usually a complex process that involves communicating with a vendor-provided service to tie the specific execution to an individual processor of that vendor.

This process includes verification that the platform where the key originated from is up-to-date and is not affected by any known and unpatched vulnerabilities. Given that complexity, remote attestation is often only used to bootstrap a more lightweight process that performs attestations more locally (like Intel DCAP [SFF20]), or even just to establish a secure channel into the enclave using symmetric authenticated encryption. The latter corresponds to implicit attestations that can be verified by a designated verifier only. A typical usage pattern for real-world enclaves is to use a single enclave to *outsource* computations, possibly with input from multiple parties. To this end, secure enclaves with remote attestation are provided by the big hyperscalers, albeit possibly with different mechanisms for remote attestation.

Formal Models for Enclaves. To analyze the security of cryptographic protocols using enclaves, numerous models have been proposed [CDL16; Sub+17; Tra+17]. A very popular model is the one proposed by Pass, Shi, and Tramèr [PST17] (subsequently used as basis for, *e.g.*, [DMM23; MK25; Bha+21]), which models (multiple) enclaves as a global ideal functionality within the Generalized UC framework [Can+07]. Considering a security notion with universal composability has the advantage that a protocol's security can be proven independently of other possibly concurrently executed protocols, while still retaining security in arbitrary protocol contexts. Considering a *global* ideal functionality is motivated by the fact that today's enclaves often share state via a common attestation infrastructure, which is not captured by a local ideal functionality within the standard UC framework [Can01]. $\mathcal{G}_{\text{att}}^{\text{PST}}$ as proposed in [PST17] allows parties to install enclaves with arbitrary code that can be accessed by the enclave's owner via inputs and outputs. Each enclave output, together with the enclave's code and a unique enclave identifier, is signed with a signing key held by $\mathcal{G}_{\text{att}}^{\text{PST}}$. The public key can be obtained by *any* party, allowing them to verify any signed output made by $\mathcal{G}_{\text{att}}^{\text{PST}}$, including those of other protocol sessions.

Compared to real-world enclaves, the life cycle modeled in [PST17] is similar. However, remote attestation is handled in a simplified way by attaching signatures on the output, enclave ID and enclave program to *every* output. In line with real-world remote attestations, these signatures share state via a common signing key and can be verified even by external parties. Details such as communication with an attestation service and revocation are not captured. Several variants of $\mathcal{G}_{\text{att}}^{\text{PST}}$ considering side channels exist, see, *e.g.*, [PST17; DMM23; MK25].

Impossibility Result: No Single Enclave Computations. Unfortunately, the model of Pass, Shi, and Tramèr [PST17] is subject to strong impossibility results. In particular, to perform an n-party computation, n enclaves are needed [PST17, Theorem 2]. This impossibility result is in stark contrast to how enclaves are used in practice, namely by *outsourcing* computations to a *single* enclave.

This impossibility result stems from the fact that the ideal functionality $\mathcal{G}_{\text{att}}^{\text{PST}}$ is *global* and designed in a way that does not allow the simulator to lie about outputs of $\mathcal{G}_{\text{att}}^{\text{PST}}$. As a consequence, the simulator has no advantage over the environment. Indeed, the simulator is generally limited to observing or

changing corrupted parties' inputs *to their own enclaves*—if a corrupted party has no enclave, then there is no direct way to extract its inputs or to generate honest enclave outputs using different inputs. At the same time, the simulator cannot lie about outputs of $\mathcal{G}_{\text{att}}^{\text{PST}}$ *at all*, because $\mathcal{G}_{\text{att}}^{\text{PST}}$ signs *every* output in a way where signatures are verifiable by *any* party, even those external to the protocol. While (honest) parties could, in principle, be instructed to drop or ignore signatures, corrupted parties will not necessarily do so and be able to detect possible manipulation by the simulator. The simulator also cannot simply present signatures that were created using a different key, because the environment could just access $\mathcal{G}_{\text{att}}^{\text{PST}}$ using a honest party from a different session and obtain the correct verification key without any possibility for the simulator to interfere. In a setting where not every party has an enclave, the simulator thus cannot hope to set up an extraction trapdoor in another way.

For equivocation, the situation is similar: Direct equivocation through $\mathcal{G}_{\text{att}}^{\text{PST}}$ is not possible and thus has to be achieved at the protocol level, which again is not possible unless every party has an enclave. To this end, the typical design pattern used in [PST17] is for enclave programs to contain an interface that returns either the real output or an arbitrary value provided as argument. Honest parties will call this interface without additional arguments, receiving the correct output. The simulator can modify a corrupted party's call to this interface to include an argument of the simulator's choice, generating a valid enclave output (that will, however, not "convince" any other party).

In order to circumvent this impossibility result, Pass, Shi, and Tramèr [PST17] propose the use of a second (global) setup, namely an *augmented common reference string*, which is used to set up an additional per-party trapdoor independent of $\mathcal{G}_{\text{att}}^{\text{PST}}$. Instead of presenting enclave signatures, holders of such a signature can, using a non-interactive witness-indistinguishable proofs (NIWIs), prove that they are either in possession of a signature of $\mathcal{G}_{\text{att}}^{\text{PST}}$ on the claimed message or know the recipient's trapdoor from the common reference string. The presented approach is limited to PID-wise corruption, requires additional coordination through a setup (which is already GUC-complete) and involves non-black-box computations through the NIWI proof. In any case, this approach does not capture how enclaves are used in the real world.

As a consequence, the model of Pass, Shi, and Tramèr [PST17], even when additionally using an augmented common reference string, is unsuitable to analyze the security of many natural protocols that should, arguably, be secure. Examples include outsourced computations using only a single enclave, a single trusted initializer that generates correlated randomness or even the naïve passive-to-active compiler where every party just delegates protocol execution to its enclave and verifies the attestation of every message received by other parties.

This issue motivates our first research question:

Is there a meaningful variant of $\mathcal{G}_{\text{att}}^{\text{PST}}$ that allows (black-box) composable general secure multi-party computation with a single enclave only?

Impossibility Result: No Global Public Verifiability. Public verifiability allows to prove the correctness of a computation's result even to third parties that have not participated in that computation. To this end, two lines of definitions exist in the literature. One approach presented, *e.g.*, in [BDO14] is to consider an *external* verifier that is given information about the protocol execution and is supposed to use this information to verify the correctness of a (claimed) computation result. In a UC-like setting where the ideal-world adversary may have unlimited influence on the used setups, it is unclear what guarantees such an external verifier (that is not given security guarantees through the executed ideal functionality) enjoys in the ideal execution. After all, the ideal-world adversary could, for example, have programmed a local random oracle such that the computation result is totally different. Thus, the security guarantees given by such an approach are unclear.

A second line of definition, also investigated, *e.g.*, in [BDO14], considers verifiers that are part of the *same* protocol session. While such verifiers should enjoy meaningful security guarantees in an ideal execution (through the ideal functionality), making the verifier part of the protocol severely limits the scope of verifications in *other* protocols that the verifier is not part of. Moreover, verification is only possible throughout the lifetime of the protocol. When the protocol and its execution have finished, verification may no longer be possible. Also, problems with respect to transferability arise. Arguably, for public verifiability, any verifying party that enjoys public verifiability should be able to transfer the evidence that allows verification to another party, which should then be able to perform the verification in the same manner. If this other party is not again part of the protocol, this is generally not possible in a secure way because it cannot access the hybrid functionalities within the protocol session in question.

Given that enclaves can attest outputs even to parties that are *outside* a protocol session, a stronger notion of public verifiability should be possible. In particular, public verifiability should be possible a) even for parties that are *not* part of the protocol session under consideration, b) after the protocol session under consideration has finished, c) in a way that evidence can be meaningfully *transferred* to other parties, who also should be able to verify this evidence, d) in a way that is sound for executions of both real and ideal protocols, e) and that provides sensible guarantees even if all parties participating in the protocol are corrupted (up to the point possible in this setting). We call public verifiability with these properties *global* and give a formal definition in Sect. 5.

At first glance, it seems that giving *any* party the ability to verify enclave outputs (tied to an enclave's code) would result in a setting where such *global public verifiability* can be easily achieved. For example, consider a coin-toss for a string of length linear in the security parameter performed by two (possibly corrupted) parties P_1 and P_2. To this end, there are (at least) two intuitive approaches using enclaves: In the first approach, parties P_1 and P_2 provide inputs (*e.g.*, randomness) to one or more enclaves which are then used to somehow compute the result (*e.g.*, by XORing the inputs). In such a setting, it is conceivable that simulation is possible by having the simulator contribute inputs such that the result matches the one from the ideal coin-toss functionality, which is how

simulation in [PST17] works. However, the resulting protocol arguably does not achieve any reasonable definition of public verifiability, as an *adversarial* entity, here the *ideal-world adversary* or simulator, is able to equivocate the result. This is not problematic within the session of the coin-toss itself, because within the session, honest parties should enjoy the security guarantees of the ideal coin-toss functionality. However, the equivocation *crosses protocol boundaries*, allowing the simulator (and thus, in GUC, also the environment), to lie to *honest* protocol parties *outside the protocol session*, which it should clearly not be able to, given that public verifiability aims to provide security guarantees even for (honest) parties that do *not* participate in the protocol.

To fix this shortcoming, a natural approach would be to use an enclave that samples the result without any external input and generates an appropriate output. Clearly, such an attested result is convincing to any external party, because even if P_1 and P_2 are corrupted, they cannot possibly influence the computation (up to a polynomial number of retries using fresh enclaves for each try). However, we cannot possibly hope to prove the GUC security of such a protocol, because the simulator has no way to reconcile the output of the coin-toss functionality (which it cannot and must not influence) with the publicly verifiable output of the enclave, which it also must not be able to influence in order for global public verifiability to make any sense.

We prove the above intuition with an impossibility result. In more detail, we show that $\mathcal{G}_{\text{att}}^{\text{PST}}$ from [PST17] cannot provide even a weak variant of global public verifiability, even if a global public bulletin board (*e.g.*, acting as a public-key infrastructure) and an augmented common reference string are provided.

Theorem 1 (informal). *There is an ideal functionality $\mathcal{F}$, such that there is no protocol π using $\mathcal{G}_{\text{att}}^{\text{PST}}$ and (optionally) $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ that both realizes $\mathcal{F}$ and is also weakly globally publicly verifiable.*

For the full theorem and definition of public verifiability see Sect. 5.

This leads to our second research question:

Can we reconcile the apparent contradiction between global public verifiability and simulation-based security, even in a setting where all parties are corrupted?

An Improved Model. We answer both questions affirmatively by presenting two variants of $\mathcal{G}_{\text{att}}^{\text{PST}}$ that jointly a) allow to analyze and prove the security of many natural protocols using a single enclave only and b) enable protocols with global public verifiability.

Compared to $\mathcal{G}_{\text{att}}^{\text{PST}}$, our proposed global ideal functionalities completely rework how attestations are handled. First of all, they are *not* added to every enclave output anymore, but must now be explicitly requested by the enclave program. We introduce *designated-verifier attestations*, which are attestations that are only valid *for a designated party within a single session*. If such an attestation is set up for a corrupted party, the simulator can now lie about the attested value. In particular, designated-verifier attestations cannot be verified by any other party.

In more detail, we present two variants of $\mathcal{G}_{\text{att}}^{\text{PST}}$. In the first variant (see Sect. 3), attestations are idealized and handled by dedicated enclave interfaces. To model the guarantees offered by globally verifiable signatures in $\mathcal{G}_{\text{att}}^{\text{PST}}$, we offer a global attestation interface that can be used by any party to check the validity of a (presumptive) enclave output and enclave code. To model designated-verifier attestations, we offer a similar interface at the granularity of single parties within single sessions (after an initial per-session registration). If a corrupted party queries our variant of $\mathcal{G}_{\text{att}}$ to verify a designated-verifier attestation, the simulator can simply lie about the output. As such attestations are tied to a party, the environment cannot query $\mathcal{G}_{\text{att}}$ for the same attestation using a different identity. Attestations are tied to an attestation identifier (AID) that has a uniformly random prefix and an adversarially chosen suffix. AIDs prevent an unintended side-channel where parties could indirectly communicate through $\mathcal{G}_{\text{att}}$ by setting up resp. checking for existing attestations without performing any additional communication whatsoever. By introducing AIDs with high entropy, such communication is not possible, except with negligible probability.

We also consider a second variant (see Sect. 6) where attestations are realized using cryptographic building blocks. For globally verifiable attestations, we keep using signatures, similar to [PST17]. For designated-verifier attestations, a verifying party first needs to perform a registration phase. This is modeled as a Diffie-Hellman key exchange where the enclave's messages are signed with the same key used for globally verifiable attestations. The exchanged key is then used to generate MAC tags that can only be verified by the designated verifier. If the verifier is corrupted, $\mathcal{G}_{\text{att}}$ hands out the exchanged key to the simulator, allowing it to subsequently fake MAC tags. We stress that the simulator remains unable to lie about signatures / global attestations generated by $\mathcal{G}_{\text{att}}$ and thus does not affect the security guarantees provided to parties in other sessions. We present a first version of $\mathcal{G}_{\text{att}}^{\text{MAC}}$ in Fig. 1, highlighting the differences to $\mathcal{G}_{\text{att}}^{\text{PST}}$ from [PST17].

Using designated-verifier attestations, per-party trapdoors for extraction and equivocation can be set up at the protocol level. For example, parties could perform a second key exchange (as part of the protocol) with an enclave for the later use with a (symmetric) authenticated encryption scheme. For a corrupted party, the simulator could perform a man-in-the-middle attack during this key exchange, allowing it to decrypt ciphertexts intended for the enclave or to produce ciphertexts (presumptively) coming from the enclave.

Using this pattern allows to decouple *explicit* attestations handled by the model and *implicit* ones by the protocol, just like in the real world, where attestations that are provided by the enclave are often costly, for example because they require the interaction with an external service or expensive public-key attestations. Compared to the combination of $\mathcal{G}_{\text{att}}^{\text{PST}}$ and $\mathcal{G}_{\text{acrs}}$, our proposed model handles the setup of simulation trapdoors in a much more fine-grained (*i.e.*, per-session versus global) way, tailored to the task at hand, namely computations using enclaves. Moreover, we have explicitly introduced different interfaces for global and designated-verifier attestations, with the former not subsuming the

$\mathcal{G}_{\text{att}}^{\text{MAC}}[\Pi, \Sigma]$ (partial)

On initialize: $T = \emptyset, O = \emptyset, V = \emptyset, \mathsf{key} = \emptyset, (vk, sk) \leftarrow \Sigma.\mathsf{KeyGen}$.

On receive `Getvk`() from some P_j: send vk to P_j.

On input `Install`(idx, prog) from some P_i:

- If P_i is honest, verify that $idx = sid$. Abort otherwise.
- Generate nonce $eid \xleftarrow{\$} \{0,1\}^\lambda$, store $T[eid] := (idx, \mathsf{prog}, \bot, P_i)$, send eid to P_i.

On input `Resume`(eid, inp) from P_i:

- Let $(\cdot, \mathsf{prog}, \mathsf{mem}, P) := T[eid]$ or ignore this input if not found or if $P_i \neq P$.
- Let $(\mathsf{out}, \mathsf{mem}') := \mathsf{prog}(\mathsf{mem}, \mathsf{inp})$ and store $O[eid] := O[eid] \cup \{\mathsf{out}\}$.
- ~~Compute $\sigma \leftarrow \Sigma.\mathsf{Sign}(sk, idx, eid, \mathsf{prog}, \mathsf{out})$.~~
- Update $T[eid] := (\cdot, \mathsf{prog}, \mathsf{mem}', P_i)$ and send ~~(~~ out ~~, σ)~~ to P_i.

On receive `Register`(eid, P_j, m) from P_i:

- If $\mathsf{key}[P_j, eid] \neq \bot$, ignore this call.
- Sample $b \xleftarrow{\$} \mathbb{Z}_p$ and store $\mathsf{key}[P_j, eid] = m^b$.
- If P_j is corrupted, send m^b to the adversary.
- Return $(g^b, \sigma \leftarrow \Sigma.\mathsf{Sign}(sk, (\text{“}\texttt{register}\text{”}, P_j, eid, (m, g^b))))$.

Enclave-Internal Operations

On receive `AddAttestation`(P, out) from a TEE with an enclave ID eid and program prog:

- Verify that $\mathsf{key}[P, eid] \neq \bot$, ignore call otherwise.
- Verify that $\mathsf{out} \in O[eid]$, ignore call otherwise.
- Let $(idx, \mathsf{prog}, \cdot, P_i) := T[eid]$.
- If $(P, idx, \mathsf{prog}, eid, \cdot, \mathsf{out})$ is not stored in V, sample $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$ and generate $aid_2 := \Pi.\mathsf{Mac}(\mathsf{key}[P, eid], (\text{“}\texttt{attest}\text{”}, idx, \mathsf{prog}, eid, aid_1, \mathsf{out}))$.
- Set $V := V \cup \{(P, idx, \mathsf{prog}, eid, aid := aid_1 || aid_2, \mathsf{out})\}$.
- Otherwise, get aid from V.
- Return aid.

On receive `AddAttestationAll`(out) from a TEE with an enclave ID eid and program prog:

- Verify that $\mathsf{out} \in O[eid]$, ignore call otherwise.
- Let $(idx, \mathsf{prog}, \cdot, P_i) := T[eid]$.
- If $(idx, \mathsf{prog}, eid, \cdot, \mathsf{out})$ is not stored in V, sample $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$ and generate $aid_2 := \Sigma.\mathsf{Sign}(sk, (\text{“}\texttt{attest}\text{”}, idx, \mathsf{prog}, eid, aid_1, \mathsf{out}))$.
- Set $V := V \cup \{(idx, \mathsf{prog}, eid, aid := aid_1 || aid_2, \mathsf{out})\}$.
- Otherwise, get aid from V.
- Return aid.

Fig. 1: Partial ideal functionality $\mathcal{G}_{\text{att}}^{\text{MAC}}$ with MAC- and signature-based attestation. Semantic changes to [PST17] are highlighted.

latter (even though, in $\mathcal{G}_{\text{att}}^{\text{MAC}}$, signatures are used to secure the key exchange for the designated-verifier attestation key). One of the main reasons to this end is the fact that the interface is different: For designated-verifier attestations, the verifier needs to be specified—it is undesirable to set up a designated-verifier attestation for every output and every party. The matter of the interface for attestation is also related to the matter of the interface for simulation. In principle, the

simulator could be allowed to generate signatures under the global signing key which are (syntactically) intended for a certain party, allowing it to, *e.g.*, perform a man-in-the-middle attack on a key exchange. However, without an explicit interface, it is unclear how an enclave program could use this exchanged key.

Unfortunately, this refinement is insufficient to enable global public verifiability. Indeed, there seems to be an inherent contradiction between simulation-based security and global public verifiability: Given the ability of a simulator, which is an adversarial entity, to equivocate results *across protocol boundaries*, how should an external party trust that a certain output is not the result of an action performed by the ideal-world adversary (which may not be benign)? This problem is even more severe when considering *global* ideal functionalities: Any power granted to the simulator in one session is also implicitly available to the environment in other sessions, allowing it to fake publicly verifiable outputs across the board.

To solve this problem, we propose a novel solution to achieve global public verifiability. While we still require the simulator to be able to extract inputs of corrupted parties, we do not require it to perform equivocation of publicly verifiable outputs (which are made through $\mathcal{G}_{\text{att}}$ to be publicly verifiable). Instead, we let the local ideal functionality $\mathcal{F}$ that we realize reconcile outputs it generates with outputs that need to be produced by $\mathcal{G}_{\text{att}}$. The corresponding interface is shown in Fig. 2. To achieve any meaningful notion of security, the ability to overwrite outputs needs to be severely restricted. Indeed, it cannot possibly hold for arbitrary ideal functionalities.

The environment could spawn a session with a different ideal functionality that lets it make arbitrary publicly verifiable outputs, rendering public verifiability meaningless again. Conversely, the overwritten outputs of $\mathcal{G}_{\text{att}}$ generated by the ideal functionality must be consistent with the real-world protocol and the enclave program used therein, which an ideal functionality generally is oblivious of.

To address these challenges, namely achieving meaningful security and consistency, we restrict the ability to overwrite outputs of $\mathcal{G}_{\text{att}}$ only to a single *generic* ideal functionality $\mathcal{F}_{\mathsf{m}f}$ that can be provably used to realize (almost) any other ideal functionality and also captures the inevitable adversarial influence, namely resets if all parties are corrupted. In more detail, we consider a shell sh around $\mathcal{F}_{\mathsf{m}f}$ which handles the communication with $\mathcal{G}_{\text{att}}$, making $\mathcal{F}_{\mathsf{m}f}$ a standard UC functionality that is agnostic of how we achieve public verifiability.

The proposed approach is reminiscent of the techniques proposed by Canetti and Chen [CC25], but needs to address the challenge that simulation must not affect security guarantees of external parties. In the setting of [CC25], this is no concern, and thus no such precautions are taken.

$\mathcal{G}_{\text{att}}^{\text{MAC}}[\Pi, \Sigma]$ (continued)

Ideal Functionality Operations

On receive $\texttt{AddAttestationAll}(eid, \mathsf{out}, \mathsf{prog})$ from an ideal functionality $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$:

- Let idx be the sid of $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$.
- Sample $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$ and generate $aid_2 \coloneqq \Sigma.\mathsf{Sign}(sk, (\text{“}\texttt{attest}\text{”}, idx, \mathsf{prog}, eid, aid_1, \mathsf{out}))$.
- Set $V \coloneqq V \cup \{(idx, \mathsf{prog}, eid, aid \coloneqq aid_1 || aid_2, \mathsf{out})\}$. // Always generate a record.
- Return aid.

Fig. 2: Continuation of $\mathcal{G}_{\text{att}}^{\text{MAC}}$ with the interface for ideal functionalities.

Interestingly, even these changes do not yet yield meaningful global public verifiability: An external verifier in the possession of attestation evidence cannot generally detect if all (claimed) parties have actually participated in the computation. In an outsourcing setting with a single enclave only, a corrupted enclave host could impersonate other parties and participate in the computation on their behalf. To protect against such attacks, we additionally use a global public-key infrastructure and only consider enclave programs that a) incorporate a public verification key of a signing scheme for each party and b) only continue once they have seen a signature for a fresh message from each party that can be verified using the programmed verification key. While enclave programs and $\mathcal{G}_{\text{att}}$ cannot check if these keys are consistent with keys provided in the public-key infrastructure, an external verifier can. Again, this is handled by the shell sh.

Protocols. Using our new enclave models, we propose protocols for composable general MPC with and without public verifiability using a single enclave. Roughly, our protocols (see Sects. 4 and 5.1) work as follows: One party that may or may not participate in the computation itself acts as *enclave host*, installing a single enclave within $\mathcal{G}_{\text{att}}$. The enclave program allows every participating party to establish a key with the enclave using a Diffie-Hellman key exchange, whose output is attested by $\mathcal{G}_{\text{att}}$ using designated-verifier attestations. (Note that if $\mathcal{G}_{\text{att}}^{\text{MAC}}$ is used, an additional protocol-external Diffie-Hellman key exchange is used to establish the per-party MAC key used for attestations.) The exchanged key is used for a secure channel between party and enclave, which is realized using authenticated encryption. After all parties have provided their input (for the current round), the desired function f is evaluated and the result returned to the enclave host, who may distribute it to the recipient. The enclave host can also set up a designated-verifier attestation for the recipient.

Within the protocol, it is the host's role to forward messages between the parties and $\mathcal{G}_{\text{att}}$. The protocol parties that want to perform the computation (which may include the host) interact with the enclave through the host by first performing a key exchange, providing the (encrypted) input to the enclave and waiting for results, which they validate through designated-verifier attestations.

Theorem 2 (informal). *Assuming a secure authenticated encryption scheme and a key exchange protocol, then there exists a protocol that realizes (almost) any ideal functionality in the presence of $\mathcal{G}_{\text{att}}$ using a single enclave.*

The above approach can be extended to provide global public verifiability. To address impersonation attacks, an additional (non-GUC-complete) public bulletin board $\mathcal{G}_{\text{reg}}$ is used, which acts as a public key infrastructure for signature keys. In a first phase, protocol parties upload their public verification key to $\mathcal{G}_{\text{reg}}$. Also, the public verification keys are included in the enclave program and thus become part of the attestation. As part of the key exchange with the enclave (as part of the enclave program), protocol parties identify themselves by providing fresh signatures that can be verified using the verification keys provided to $\mathcal{G}_{\text{reg}}$. While enclaves themselves cannot access $\mathcal{G}_{\text{reg}}$, any verifying party can and subsequently check if the hard-coded keys in the enclave match those of $\mathcal{G}_{\text{reg}}$.

Theorem 3 (informal). *Assuming an authenticated encryption scheme and a key exchange protocol, then there exists a protocol that realizes (almost) any ideal functionality in the presence of $\mathcal{G}_{\text{att}}$ and $\mathcal{G}_{\text{reg}}$ with global public verifiability.*

In the above theorems, the attestations performed by $\mathcal{G}_{\text{att}}$ are information theoretic (in contrast to $\mathcal{G}_{\text{att}}^{\text{MAC}}$, which uses cryptographic building blocks). We thus prove that $\mathcal{G}_{\text{att}}$ can be realized using this variant $\mathcal{G}_{\text{att}}^{\text{MAC}}$:

Theorem 4 (informal). *Assuming the existence of a signature scheme, a message authentication code and a protocol for key exchange, $\mathcal{G}_{\text{att}}$ can be realized using $\mathcal{G}_{\text{att}}^{\text{MAC}}$.*

Relation to Real-World Enclaves. In comparison to previous models, our proposed models can be used to analyze and prove the security of single-enclave protocols that can be implemented with real-world enclaves. A notable exception is side channels, which we have not chosen to model. However, we stress that side channels can also be modeled through appropriate enclave programs, as long as the cryptographic operations baked into the model are possible in a side channel-free manner, which is plausible [DMM23].

Both of our models simplify attestations compared to real-world enclaves, which is in line with existing models. This simplification is intended because real-world attestation protocols are very complex and product-specific. However, by modularizing attestations and not requiring them for every output, our models close the gap to real-world enclaves, where attestations explicitly need to be requested. With $\mathcal{G}_{\text{att}}$, we propose a model that uses idealized attestations but allows adversarially specified AIDs. These AIDs can be used to bridge the syntactical gap between idealized attestations and real-world ones by incorporating real-world attestation messages (*e.g.*, DCAP signatures).

1.1 Related Work

Formal Models for Enclaves. The first formal model for attested execution in the UC framework was given by Pass *et al.* [PST17]. This model provides

universally composable multi-party computation if all parties host an enclave. To distinguish the different versions of $\mathcal{G}_{\text{att}}$, we call the one by Pass *et al.* $\mathcal{G}_{\text{att}}^{\text{PST}}$.

To capture the variety of different enclave models found in the literature, Martinico and Kohlweiss [MK25] present a modular framework in UC for definitions of secure enclaves based on $\mathcal{G}_{\text{att}}^{\text{PST}}$ that characterizes capabilities of trusted hardware by the trusted features available to the enclave, the set of possible attacks on an enclave and the content of attestation signatures. They define different ideal modular functionalities that represent different existing variants of enclave models and existing attacks on real world enclaves, *e.g.*, different leakage models or attestation mechanisms. For example, they relax the attestation mechanism with respect to $\mathcal{G}_{\text{att}}^{\text{PST}}$ to allow the adversary to decide upon a signature algorithm and a public key. In this work, we focus on achieving properties like global public verifiability and single-enclave computations. However, the framework of [MK25] cannot be used as-is to capture the proposed modifications. In particular, the framework is unsuitable to capture designated-verifier attestations as well as the proposed modifications that enable global public verifiability, as the interfaces of this framework are fixed. For these reasons, we directly propose variants of $\mathcal{G}_{\text{att}}$ and leave appropriate modifications to [MK25] for future work.

Outside the UC framework, several works have proposed stand-alone models of secure enclaves [Sub+17; Fis+17]. We consider these approaches out of scope as they do not consider composability.

Public Verifiability. A notion for public verifiability in a general setting is, for example, considered by Baum, Damgård, and Orlandi [BDO14]. In their work, public verifiability is called *auditable correctness* and defined for circuits. Like this work, they want to give guarantees to the auditor even in the case where all protocol participants are corrupted. They also define an ideal functionality for auditable MPC. However, they do not use this functionality in their definition of the auditable correctness of MPC protocols and limit themselves to deterministic functionalities. To achieve auditable correctness, Baum *et al.* use a local, programmable random oracle in their construction, which allows the simulator to lie to an honest auditor. Additionally, they require ideal functionalities for a bulletin board, key generation, and commitments as a setup.

In the above approach, the security guarantees given to *external* verifiers (to the point they are external—[BDO14] also consider verifiers that are part of the protocol) are different from standard simulation-based security, where honest parties are *not* affected by whatever the simulator does—after all, their outputs are generated by an ideal functionality where the simulator typically only has very little influence. To the extent an external verifier uses artifacts of the (supposedly) real-world protocol execution (*e.g.*, protocol transcripts posted on a bulletin board or random oracle images), these are possibly affected by the simulator in the ideal execution. Thus, the simulation crosses the protocol boundary and the simulator is able to *lie to these honest external verifiers*. Given that the simulator is the ideal-world adversary, external verifiers suddenly do not enjoy security guarantees anymore in the ideal execution *at all*!

In a setting where global ideal functionalities are used, the consequences are even more severe: Any capability given to the simulator in one session is implicitly given to the environment in other sessions. So even in the proof where the simulator is "well-behaved", the environment could abuse its power in another session. Consequently, for example NIZKs using a global setup that are transferable (and thus publicly verifiable) may not make any sense at all, given that the environment could possibly generate proofs for false statements that are accepted by verifiers outside the protocol.

In their work written concurrently with ours, Baum and Zok [BZ26] improve upon [BDO14] by using Oblivious Linear Evaluation as an assumption instead of lattice primitives together with the discrete logarithm problem. Further, they do not require a common reference string as a setup anymore. However, they do not touch upon the limitations described above. In particular, they still use a global, programmable random oracle. While they use the global random oracle functionality proposed by Canetti and Chen [CC25], which can only be programmed by ideal functionalities, they define a macro for each of their ideal functionalities. This allows the simulator to choose how the oracle should be programmed. [CC25] does not consider public verifiability, which poses additional challenges beyond communication between local and global ideal functionalities.

Baum, David, and Dowsley [BDD22] discuss the challenges of achieving public verifiability in UC. Their solution is to use an input-aware "über simulator" to solve issues with simulatability. They state that "über simulators" can be constructed easily when fixed or random inputs are used in the simulation strategy of the security proof. For other types of protocols, the "über simulator" would have to be defined additionally. This special type of simulator receives the honest parties' inputs and the randomness tape of the ideal functionality. Since this is an extension of the UC simulator, security guarantees become less clear.

Rausch *et al.* [RHK25] consider verifiability for e-voting. In contrast to our notion of global public verifiability, they only consider verifiers that are part of the protocol. Similarly, Baldimtsi *et al.* [Bal+20] consider end-to-end verifiable MPC, where the verifier tries to convince other protocol participants. They model their construction in the extended UC framework with super-polynomial helpers.

A property related to public verifiability is accountability. Graf, Küsters, and Rausch [GKR23] propose an accountability framework for UC, which models the identification of misbehaving protocol participants with the help of ideal (public) judges. Since the goal of accountability is different from public verifiability, we do not consider this notion any further.

In the light of the above, we propose (and realize) a stronger variant of public verifiability, which we call *global*, where even external verifiers that are not part of the protocol session enjoy essentially ideal security guarantees. We achieve this by not requiring (and allowing) the simulator to perform equivocations across protocol boundaries. Instead, we delegate this task to the ideal functionality in a way that is sound. This is much stronger than what is previously achieved and what we believe can be otherwise achieved using global ideal functionalities.

1.2 Outline

The paper starts with a presentation of the preliminaries in Sect. 2 and continues with a presentation of our novel global ideal functionality in Sect. 3. In Sect. 4, we present a protocol for composable general MPC using our new model. Global public verifiability is discussed in Sect. 5, including the corresponding protocols, definitions and a proof of our impossibility result. In Sect. 6, we propose a variant of our model where attestation is performed using cryptographic building blocks.

2 Preliminaries

This section covers the notation used in this paper, as well as a short introduction to the UC framework [Can01] and Universal Composability with Global Subroutines (UCGS) [Bad+20]. We give a short introduction to hardware enclaves. We defer the definition of INT-CTXT to App. B.

2.1 Notation

Let $\kappa \in \mathbb{N}$ denote the security parameter. Let $x \leftarrow X$ denote that some probabilistic algorithm X returns x. Let $y \xleftarrow{\$} Y$ denote that y is sampled uniformly at random from some set Y. Let $\mathsf{negl}(\kappa)$ denote an unspecified negligible function in κ. Let $\perp$ denote a null value. Let g be a generator of a cyclic group $\mathbb{G}$ of prime order p. Let $\mathbb{Z}_p$ denote the set of integers modulo p. Let $\mathcal{K}$ denote the key space for symmetric encryption keys.

2.2 The Universal Composability Framework

In the following, we give a brief overview of the Universal Composability (UC) security notion [Can01; Can20; Bad+20]. The following, as well as the overview of UCGS, is adapted from [DMM23].

In the UC framework, security is defined by the indistinguishability of two experiments: the *ideal experiment* and the *real experiment*. In the ideal experiment, the task at hand is carried out by dummy parties with the help of an ideal incorruptible entity—called the ideal functionality $\mathcal{F}$. In the real experiment, the parties execute a protocol π to solve the prescribed task themselves. A protocol π is said to be a (secure) *realization* of $\mathcal{F}$ if no PPT machine $\mathcal{Z}$, called the *environment*, can distinguish between these two experiments. In contrast to previous simulation-based notions, indistinguishability must not only hold after the protocol execution has completed, but even if the environment $\mathcal{Z}$—acting as the *interactive* distinguisher—takes part in the experiment, orchestrates all adversarial attacks, gives input to the parties running the challenge protocol, receives the parties' output and observes the communication during the whole protocol execution.

Further details about the UC experiment, protocol composition, and UC related specifics of this paper can be found in App. A. We mention two specific details herebecause we use them multiple times throughout our work:

- Each *interactive Turing machine instance (ITI)* is uniquely defined by its *extended identity* (sid, μ, pid). It consists of the session identifier sid, the code μ and the party identifier pid.
- A *structured ITM* is an interactive Turing machine that consists of a *body* and a *shell*. Both behave like separate ITMs, except that the shell can interact with other ITMs as the whole structured ITM and has read and write access to all tapes of the body, whereas the body can only interact with the shell.

2.3 Universal Composition with Global Subroutines

Initially, UC security was not designed to deal with *global ideal functionalities*, *i.e.* ideal functionalities that are used by multiple protocols. Generalized UC (GUC) security [Can+07] extended UC security to capture security in the presence of *global subroutines* such as global ideal functionalities. Unfortunately, GUC security suffers from several technical problems [Bad+20].

Consequently, UC security has been extended to allow the modeling of global ideal functionalities within UC security [Bad+20; Can20]. In more detail, the *Universal Composition with Global Subroutines* (UCGS) formalism captures the execution joint of a protocol π (as in UC) and a global ideal functionality $\mathcal{G}$ within a single so-called *management protocol* $\mathsf{M}[\pi, \text{IDEAL}(\mathcal{G})]$. As $\mathsf{M}[\cdot]$ is a UC protocol, this formalism recovers all properties of UC security.

We restate the definition of UC emulation with global subroutines.

Definition 1. *UC emulation with global subroutines [Bad+20]* Let π, ϕ and γ be protocols. We say that π ξ-UC-emulates ϕ in the presence of γ if protocol $\mathsf{M}[\pi, \gamma]$ ξ-UC-emulates protocol $\mathsf{M}[\phi, \gamma]$.

In the following, we will leave out the predicate ξ when we do not impose any restrictions on party identities.

2.4 Hardware Enclaves

Different security hardware is sold under the term *enclave*. This reaches from hardware executing only a specific program (*e.g.* HSMs, secure elements), over hardware mechanisms that isolate fixed parts of the firmware (like Arm Trust Zone), to mechanisms that allow running arbitrary programs with remote attestation, like Intel SGX, Intel TDX and AMD SEV-SNP. The latter kind are the enclaves that we want to model in this work. We take Intel SGX as a representative for this group.

From an interface level (*e.g.* with Intel SGX), the user passes an arbitrary program to the platform that is installed as an enclave, where the host can call any of the interfaces that this program has. Similarly, the host mediates access to (almost all) system resources the enclave wants to access, with CPU instructions (including access to CPU-generated randomness) and allocated memory being a notable exception. To initiate a remote attestation, the host application invokes library functions both in the host program and, through an enclave function,

inside the enclave. The enclave program defines what data is to be included into the attestation. Finally, this process produces some binary artifact. In case of SGX DCAP, this artifact can then be checked *locally* together with an artifact from the bootstrapping process for a match on some enclave program and user data. Attestation evidence from Intel SGX and Intel TDX is verifiable by everyone with access to the corresponding Intel services. This allows a relying party to ensure that the output was produced by the expected program running inside the expected hardware enclave.

The flexibility of running arbitrary code also bears pitfalls. If not written carefully, enclave programs can exhibit side channels, that break the initial security assumption. In case of SGX, side channels like timing, power consumption or memory access patterns are accepted and placed as a burden on the designer of the enclave code. Given the nature of side channels, those are usually not explicitly and exhaustively specified.

3 Enclave Model

In this section we present our enclave model. First, we give a high-level explanation of the model including the most notable differences between our ideal functionality and $\mathcal{G}_{\text{att}}^{\text{PST}}$ by Pass, Shi, and Tramèr [PST17]. Then a detailed presentation of the enclave model follows in Fig. 3. In Sect. 6, we give an example how the general model can be instantiated using digital signatures and MACs.

To improve the readability of $\mathcal{G}_{\text{att}}$ and the protocols, we refer to the extended identity $(sid, \mu, pid)_P$ of party P simply with P. We keep consistent with the notation of $\mathcal{G}_{\text{att}}^{\text{PST}}$ by assuming unique session identifiers when we write sid, which could easily be realized by including the code as well. Furthermore, we omit session identifiers in protocol messages.

3.1 Differences Between $\mathcal{G}_{\text{att}}$ and $\mathcal{G}_{\text{att}}^{\text{PST}}$

The most notable difference between our ideal functionality and $\mathcal{G}_{\text{att}}^{\text{PST}}$ are:

- $\mathcal{G}_{\text{att}}^{\text{PST}}$ implements attestations by outputting signatures. In our model, we use interfaces in the ideal functionality that parties can query to verify the validity of attestations. We therefore separate policy and mechanism in our model, where Pass *et al.* have signatures baked into their ideal functionality.
- In $\mathcal{G}_{\text{att}}^{\text{PST}}$, an attestation can, in principle, be verified by all parties. In our model we allow attestations to be either global (being, in principle, verifiable by all parties) or designated-verifier (verifiable only by one designated party). This change allows us to effectively use a *single* enclave as a trusted third party (see Sect. 4), which is not possible with $\mathcal{G}_{\text{att}}^{\text{PST}}$ due to an impossibility result which requires an enclave for every party or an additional setup.
- In $\mathcal{G}_{\text{att}}^{\text{PST}}$, a signature is attached to every output made by the enclave, effectively creating a remote attestation for every output of the enclave. In our model, $\mathcal{G}_{\text{att}}$ has additional interfaces, where attestations are added to outputs. These interfaces can only be called from within an enclave program

$\mathcal{G}_{\text{att}}$

On initialize: $T = \emptyset, O = \emptyset, V = \emptyset, k = \emptyset, sk = \bot$. Send `Initialize` to $\mathcal{A}$. If $\mathcal{A}$ returns two algorithms $\Pi = (\mathsf{KeyGen}, \mathsf{Mac})$ and $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and sk, store them. If $\mathcal{A}$ returns $\bot$, set $\Pi.\mathsf{Mac} = \Sigma.\mathsf{Sign} = \Pi.\mathsf{KeyGen} = \Sigma.\mathsf{KeyGen} = 0^\kappa$ to the algorithm that always returns the all-zero bitstring of length κ.

On input `Install`(idx, prog) from some P_i:

- If P_i is honest, verify that $idx = sid$. Abort otherwise.
- Generate nonce $eid \xleftarrow{\$} \{0,1\}^\kappa$, store $T[eid] := (idx, \mathsf{prog}, \bot, P_i)$, send eid to P_i.

On input `Resume`(eid, inp) from P_i:

- Let $(idx, \mathsf{prog}, \mathsf{mem}, P) = T[eid]$. Ignore this input if not found or if $P_i \neq P$.
- Let $(\mathsf{out}, \mathsf{mem}') := \mathsf{prog}(\mathsf{mem}, \mathsf{inp})$ and store $O[eid] := O[eid] \cup \{\mathsf{out}\}$.
- Update $T[eid] := (idx, \mathsf{prog}, \mathsf{mem}', P_i)$ and send out to P_i.

On receive `VerifyAttestation`$(idx, \mathsf{prog}, eid, aid, \mathsf{out})$ from a party P:

- Return 1 if $(P, idx, \mathsf{prog}, eid, aid, \mathsf{out}) \in V$. Return 0 otherwise.

On receive `VerifyAttestationAll`$(idx, \mathsf{prog}, eid, aid, \mathsf{out})$ from a party P:

- Return 1 if $(idx, \mathsf{prog}, eid, aid, \mathsf{out}) \in V$. Return 0 otherwise.

Enclave-Internal Operations

On receive `AddAttestation`(P, out) from a TEE with EID eid and program prog:

- Verify that $\mathsf{out} \in O[eid]$, ignore call otherwise.
- Let $(idx, \mathsf{prog}, \cdot, P_i) := T[eid]$.
- If $(P, idx, \mathsf{prog}, eid, aid, \mathsf{out}) \in V$ for some aid, return aid.
- Sample $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$.
- If $k[P, eid] = \bot$, send (eid, P_i) to $\mathcal{A}$ and request k for P from $\mathcal{A}$. If P is corrupted and $\mathcal{A}$ returns k, let $k[P, eid] := k$, otherwise sample $k[P, eid] \leftarrow \Pi.\mathsf{KeyGen}(1^\kappa)$.
- Let $aid_2 := \Pi.\mathsf{Mac}(k[P, eid], (\text{“}\texttt{attest}\text{”}, idx, \mathsf{prog}, eid, aid_1, \mathsf{out}))$.
- Set $V := V \cup \{(P, idx, \mathsf{prog}, eid, aid := aid_1 || aid_2, \mathsf{out})\}$.
- Return aid.

On receive `AddAttestationAll`(out) from a TEE with EID eid and program prog:

- Verify that $\mathsf{out} \in O[eid]$, ignore call otherwise.
- Let $(idx, \mathsf{prog}, \cdot, P_i) := T[eid]$.
- If $(idx, \mathsf{prog}, eid, aid, \mathsf{out}) \in V$ for some aid, return aid.
- Sample $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$.
- If $sk = \bot$, sample $(sk, \cdot) \leftarrow \Sigma.\mathsf{KeyGen}(1^\kappa)$.
- Let $aid_2 := \Sigma.\mathsf{Sign}(sk, (\text{“}\texttt{attest}\text{”}, idx, \mathsf{prog}, eid, aid_1, \mathsf{out}))$.
- Set $V := V \cup \{(idx, \mathsf{prog}, eid, aid := aid_1 || aid_2, \mathsf{out})\}$.
- Return aid.

Ideal Functionality Operations

On receive `AddAttestationAll`$(eid, \mathsf{out}, \mathsf{prog})$ from an ideal functionality $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$:

- Let idx be the sid of $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$.
- If $(idx, \mathsf{prog}, eid, aid, \mathsf{out}) \in V$ for some aid, return aid.
- Sample $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$.
- If $sk = \bot$, sample $(sk, \cdot) \leftarrow \Sigma.\mathsf{KeyGen}(1^\kappa)$.
- Let $aid_2 \leftarrow \Sigma.\mathsf{Sign}(sk, (\text{“}\texttt{attest}\text{”}, idx, \mathsf{prog}, eid, aid_1, \mathsf{out}))$.
- Set $V := V \cup \{(idx, \mathsf{prog}, eid, aid := aid_1 || aid_2, \mathsf{out})\}$.
- Return aid.

Fig. 3: Ideal functionality $\mathcal{G}_{\text{att}}$

(or by an ideal functionality) and no attestation is created, unless the enclave program calls this interface. An enclave output can either have no attestation, one or multiple designated verifier attestations, and/or one or more publicly verifiable ones. Having the option of no attestation for an enclave output allows parties to deny their participation in the protocol. The non-deniability issue identified by Pass *et al.* is strongly linked to their impossibility result [PST17]. If not all parties host their own enclaves in their model, the adversary can prove to outsiders that some honest party participated in the protocol, due to the global nature of attestations in $\mathcal{G}_{\text{att}}^{\text{PST}}$. Without an attestation, the output cannot be linked to an enclave host. To still guarantee authenticity and integrity of outputs without attestations, in the protocol (as done as a first step in our protocol in Constr. 1), parties can perform an attested key exchange with the enclave to establish a secure channel. This is similar to how remote attestation and authenticated encryption are used in hardware enclaves such as Intel SGX. On a side note, adding (publicly verifiable) attestations to every output would lead to simulation problems or would lead to the impossibility result by Pass, Shi, and Tramèr. Attestations become interesting again once a third party should be convinced of the output having been computed by the enclave. To allow protocols to add attestations as needed, they must be added in an extra step to an output.

- $\mathcal{G}_{\text{att}}^{\text{PST}}$ is a global ideal functionality, and any party can request the public key necessary to verify an attestation. Even though the signatures used in $\mathcal{G}_{\text{att}}^{\text{PST}}$ seem to be publicly verifiable, for simulation to work out for protocols using $\mathcal{G}_{\text{att}}^{\text{PST}}$, the enclave programs need to have a backdoor for the simulator which allows any value to be attested, effectively making attestations meaningless in a public-verifiability context. As we will see in Sect. 5, using our model of $\mathcal{G}_{\text{att}}$, we can achieve meaningful (global) public verifiability.
- Similar to $\mathcal{G}_{\text{att}}^{\text{PST}}$, using $\mathcal{G}_{\text{att}}$, the same impossibility result with regards to fairness applies [PST17, Theorem 6].

3.2 Interfaces of $\mathcal{G}_{\text{att}}$

Figure 3 shows our global attested execution functionality $\mathcal{G}_{\text{att}}$. In the following, a detailed overview of the interfaces is given. On initialization, $\mathcal{G}_{\text{att}}$ creates three empty lists. T stores information about every enclave with identifier *eid* installed by the host P_i. A similar list is also used in $\mathcal{G}_{\text{att}}^{\text{PST}}$. O stores outputs of enclaves. V keeps track of the verifiable outputs and the parties that are allowed to verify them. The adversary additionally has the ability to upload attestation algorithms that are intended for the creation of attestations. Jumping ahead, this is a technical necessity to allow simulation of $\mathcal{G}_{\text{att}}$ (see Sect. 6). To realize the attestation in $\mathcal{G}_{\text{att}}$ using cryptographic primitives, this backdoor allows the simulator to upload the algorithms that are used in the real protocol. The choices of Π and Σ do not affect the security of $\mathcal{G}_{\text{att}}$. The verification is performed via the verification interfaces and cannot be forged.

Installation. Enclaves can be installed by all parties using the `Install` interface. While in $\mathcal{G}_{\text{att}}^{\text{PST}}$, the enclave hosts are restricted to parties in a special registry reg, we argue that nowadays, TEEs are ubiquitous. In particular, they are offered by hyperscalers, parts of (server) processors or smartphones. Therefore, we do not want to limit the possible hosts to parties listed in such a registry. One technical consequence of this choice is that in the model presented in this paper, the simulator is able to install enclaves via the installation interface as well.

Upon installation, an index idx, the enclave program prog, the so far empty enclave memory and the extended identity of the enclave host are stored in T. A random bitstring is sampled as the eid, just as in $\mathcal{G}_{\text{att}}^{\text{PST}}$. Honest parties are bound to input their session identifier as index. If they do not use the session identifier of their extended identity, $\mathcal{G}_{\text{att}}$ does not install the enclave and aborts the protocol execution. This mechanism prevents chosen protocol attacks and was already introduced in $\mathcal{G}_{\text{att}}^{\text{PST}}$. It serves the same purpose for $\mathcal{G}_{\text{att}}^{\text{PST}}$, as it does for our $\mathcal{G}_{\text{att}}$. For more details of the implications of this, see App. C.

Resume. `Resume` allows parties to give input to enclaves. Depending on the input inp, the output out is computed, and the enclave memory mem is updated. Unlike $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{att}}$ does not provide a signature (or attestation) for the output.

Adding an Attestation. An attestation consists of two parts, the randomly drawn aid_1, and aid_2, which is created using an authentication algorithm. To add attestations to outputs, there are two kinds of interfaces. First, `AddAttestation` is used for designated-verifier attestations and can only be called from within an enclave program. This limits the output values and parties that attestations are issued for to those specified in the enclave program. When calling the interface, an enclave can specify the output that is supposed to be attested together with the party that will be allowed to verify the attestation. The interface outputs an attestation ID aid, which must be specified upon verification and can be seen as a replacement for the signature. In other words, for a third party to be able to verify an attestation, the AID aid must be transferred to this party. If this transfer were not necessary, the verifier could learn additional information (*i.e.* at which point in time the verification succeeds) without ever interacting with protocol participants or any other party aside from $\mathcal{G}_{\text{att}}$. By sending the AID, this information transfer is made explicit. An AID is also similar to a signature in the sense that it can be used as an input for other computations. As it is possible to, *e.g.*, commit to a signature, one can also commit to an AID.

Secondly, `AddAttestationAll` can be called within enclave programs and by other ideal functionalities. This interface outputs publicly verifiable attestations. `AddAttestationAll` is available within enclave programs and works in a similar way as `AddAttestation`, but no verifier party is specified. We do not allow enclave hosts to add publicly verifiable attestations to outputs, since a corrupted host might create attestations which honest verifiers might later want to verify. This would lead to issues with simulation, since the simulator should not be allowed to lie to honest verifiers from different sessions. Every party in possession

of *aid*, *sid*, the enclave ID and program, and the (presumptive) output value is later able to verify the attestation, even if it belongs to a different session itself.

The `AddAttestationAll` interface for ideal functionalities serves an additional purpose. Assume a probabilistic function, *e.g.*, a coin-toss, is performed. In the real world, the enclave program samples a random bit. When replacing the protocol with an ideal coin-toss functionality $\mathcal{F}_{\mathrm{CT}}$ in the ideal world, $\mathcal{F}_{\mathrm{CT}}$ would sample a bit as well, matching the bit sampled by the enclave only with probability $1/2$. When not considering public verifiability, this would not be a problem, since the simulator could take care of inconsistencies. However, with public verifiability, there might be honest parties from different sessions trying to verify outputs, which the simulator should not lie to. To keep the publicly verifiable attestations and the outputs of the ideal functionality consistent, we instead allow certain well-behaving ideal functionalities to create attestations via `AddAttestationAll` for any values of their choice. Even if the enclave with ID *eid* has not been installed, the ideal functionality can create attestation records for this *eid*. Therefore, in addition to the output value that is supposed to be attested, the ideal functionality provides an *eid* and program prog to this interface. Looking ahead, this program is the canonical enclave program for $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$, $\mathsf{prog}_f[\{(P, vk_P)\}_{pid_P \in \mathcal{P}}]$, which is parameterized with the extended identities and verification keys of the parties participating in the protocol.

Every AID *aid* consists of the concatenation of two parts, aid_1 and aid_2. If the adversary uploaded attestation algorithms Π and Σ and a key k_2 in the initialization phase, in all three interfaces, aid_1 is drawn randomly and aid_2 is created using the respective attestation algorithms $\Pi.\mathsf{Mac}$ and $\Sigma.\mathsf{Sign}$, signing aid_1 together with the output value and some identifiers. In `AddAttestationAll`, $\Sigma.\mathsf{Sign}$ together with k_2 are used to create the *aid*. In `AddAttestation`, $\mathcal{G}_{\mathrm{att}}$ asks the adversary to provide a MAC key for each party an attestation is created for. Then, $\Pi.\mathsf{Mac}$ can be used with the provided key to create aid_2. This key is stored for further attestation requests for this party. We emphasize that, with $\mathcal{G}_{\mathrm{att}}$, the verification of attestations is always performed by $\mathcal{G}_{\mathrm{att}}$ itself and is independent of the security of the underlying cryptographic building blocks. If the adversary did not upload attestation algorithms, aid_2 is a zero-bitstring. The choice of adding the computational-theoretic aid_2 to attestations has a technical reason which will be used in Sect. 6.

$\mathcal{G}_{\mathrm{att}}$ stores AIDs that have been output before. Therefore, if an AID is requested multiple times for the same output in the publicly verifiable case or for the same output and verifier in the designated-verifier case, the AID output by $\mathcal{G}_{\mathrm{att}}$ will be the same as before. Letting the adversary provide attestation algorithms and keys does not pose authenticity issues of attestations. Honest parties only use `VerifyAttestation(All)` to verify attestations. (Fake) attestations created by the adversary will not be verified by $\mathcal{G}_{\mathrm{att}}$.

Verifying an Attestation. Like the attestation interfaces, there are two verification interfaces: one for designated-verifier attestations, which is called `VerifyAttestation`, and one for publicly verifiable ones,

`VerifyAttestationAll`. When a verifier provides the necessary information to $\mathcal{G}_{\text{att}}$, it is checked whether the information matches an entry in the verification list V. In the designated-verifier case, the identity of the verifier must be stored in the list together with the entry of the specified output. If a matching entry can be found, the interface returns 1, otherwise 0.

Remark 1. The motivation of defining the AIDs as described above is to a) allow to later on show that $\mathcal{G}_{\text{att}}$ can be realized with $\mathcal{G}_{\text{att}}^{\text{MAC}}$ where attestations are performed using cryptographic building blocks, while at the same time, b) prevent an adversary from specifying algorithms for AIDs that make them predictable, establishing an unintended communication channel. The latter is motivated to exclude pathological protocols from being provable as secure.

We stress that other designs for AIDs within $\mathcal{G}_{\text{att}}$ are possible. Like in [PST17], one could parameterize $\mathcal{G}_{\text{att}}$ with an EUF-CMA-secure signature scheme and use signatures as AIDs without a random prefix. Similarly, one could just use uniformly random AIDs without any adversarial or algorithmic influence.

Remark 2. We chose to model the verification of attestations as immediate, without the adversary being able to delay the result. First, this matches instantiations with cryptographic primitives like signatures or MACs, which can be verified locally. Secondly, the adversary already has the opportunity to interfere in the verification process by delaying the message to the verifier containing the AID.

4 General MPC with a Single Enclave

We will now use our enclave model $\mathcal{G}_{\text{att}}$ for general multi-party computation via the protocol π_f. With $\mathcal{G}_{\text{att}}$, it is possible to use a single enclave as a trusted third party, effectively allowing it to react to messages from parties (delivered by the enclave host as a proxy), perform a computation, and send outputs to a party (again, by using the enclave host as a proxy), just as an ideal functionality would. To model the security of this approach, we consider the ideal functionality $\mathcal{F}_f$.

Definition 2. *The Ideal Functionality.* Let $\mathcal{P}$ be a set of parties. Let t be a polynomial and let $f\colon \{0,1\}^* \times \{0,1\}^{t(\kappa)} \times \mathcal{P} \times \{0,1\}^* \mapsto \{0,1\}^* \times (\mathcal{P} \cup \{\bot\}) \times \{0,1\}^*$ be an efficiently computable function, then we define the ideal functionality $\mathcal{F}_f$ as follows:

$\mathcal{F}_f$

- On first invocation, set state $\coloneqq \varepsilon$.
- On input inp from party P with $pid_P \in \mathcal{P}$,
  - Notify the adversary about this input by sending $(|\mathsf{inp}|, P)$ and wait for the adversary to acknowledge this.
  - Sample $r \xleftarrow{\$} \{0,1\}^{t(\kappa)}$.
  - Compute $(\text{state}', P', \mathsf{out}) \coloneqq f(\text{state}, r, P, \mathsf{inp})$.
  - Set state $\coloneqq$ state$'$.

- If $P' \neq \bot$, send a public delayed output out to P'.
- If $P' = \bot$, send ("`output`") to $\mathcal{A}$.

– Ignore all inputs from other parties.

Note that by choosing f accordingly, $\mathcal{F}_f$ can implement any well-formed local ideal functionality, where inputs and outputs can be delayed by the adversary. By "well-formed", we mean the definition in [Can+02], where the core of the ideal functionality is independent of which parties are corrupted. Examples for such functionalities are commitments, oblivious transfer, and secure function evaluation. See App. D for more concrete examples. This approach allows us to later generically realize any such ideal functionality with a protocol by instantiating f accordingly.

Also note that in our ideal functionality $\mathcal{F}_f$, there is only a single input interface, but these inputs can consist of multiple parts. If f is, for example, instantiated as a commitment, an example input could be $\mathsf{inp} = (\text{"}\texttt{commit}\text{"}, m)$.

Using our $\mathcal{G}_{\text{att}}$, we can now construct a protocol realizing $\mathcal{F}_f$ while using only a single enclave, therefore circumventing the impossibility result of [PST17]. Our protocol π_f (Constr. 1) uses an enclave as a trusted party, where communication is done via the party hosting the enclave. Essentially, our protocol works using the following steps:

1. One party installs the enclave.
2. Each party performs an attested key exchange with the enclave.
3. Parties ensure they have exchanged the key with the *same* enclave instance by exchanging the enclave ID *eid* of the enclave they have performed the key exchange with.
4. Parties send their inputs to the enclave, encrypted using the previously established key, and receive their outputs from the enclave.

Construction 1 *The protocol π_f.* Let $\mathcal{P}$ be a set of parties and let H be a party with an enclave. ($pid_H \in \mathcal{P}$ is not necessary. If $pid_H \notin \mathcal{P}$, we are in the case, where a third party has the enclave.) Let t be a polynomial and let $f\colon \{0,1\}^* \times \{0,1\}^{t(\kappa)} \times \mathcal{P} \times \{0,1\}^* \mapsto \{0,1\}^* \times (\mathcal{P} \cup \{\bot\}) \times \{0,1\}^*$ be an efficiently computable function. We define two parts π_f^{comp} and π_f^{host} of the protocol π_f, as well as an enclave program prog_f, that is used in the protocol. π_f^{comp} is executed by all parties in $\mathcal{P}$, while π_f^{host} is executed by the party hosting the enclave. If $pid_H \in \mathcal{P}$, H executes both protocols simultaneously, potentially sending messages to itself from π_f^{comp} to π_f^{host} or vice versa.

Enclave program prog_f

Initialize state $= \varepsilon, \mathsf{L} = \emptyset$

On input ("`keyExchange`", P, X_P): sample $y_P \xleftarrow{\$} \mathbb{Z}_p$, store X_P and $k_P = X_P^{y_P}$, and output (X_P, g^{y_P}).

On input ("attest", P): If k_P and y_P have been previously stored, compute $aid_P \leftarrow \texttt{AddAttestation}(P, (X_P, g^{y_P}))$, output aid_P.
On input ("input", P, c_{inp}): Verify $c_{\text{inp}} \notin \mathsf{L}$, abort otherwise. Set $\mathsf{L} := \mathsf{L} \cup \{c_{\text{inp}}\}$. Compute $\mathsf{inp} := \mathsf{AE.Dec}(k_P, c_{\text{inp}})$. Ignore input if $\mathsf{inp} = \bot$. Sample $r \xleftarrow{\$} \{0,1\}^{t(\kappa)}$. Compute $(\text{state}', P', \mathsf{out}) = f(\text{state}, r, P, \mathsf{inp})$. Store $\text{state} := \text{state}'$. Draw $\nu \xleftarrow{\$} \{0,1\}^{\kappa}$. Output (ν, P', out) and store it as the latest output.
On input("attestOutput"): For the latest output (ν, P', out), verify that $P' \neq \bot$. Then, compute $aid \leftarrow \texttt{AddAttestation}(P', (\nu, P', \mathsf{out}))$, output aid and set the latest stored output to $\bot$.

Protocol π_f^{host} for the enclave host H

On first activation with identifier sid:
- For each party P with $pid_P \in \mathcal{P}$:
  - Await ("keyExchange", X_P) from each party P.
- Call $\mathcal{G}_{\text{att}}.\texttt{Install}(sid, \mathsf{prog}_f)$ and receive and store eid.
- For each party P with $pid_P \in \mathcal{P}$:
  - Call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{"keyExchange"}, P, X_P))$ and receive (X_P, Y_P) from $\mathcal{G}_{\text{att}}$.
  - Call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{"attest"}, P))$ and receive aid from $\mathcal{G}_{\text{att}}$.
  - Send ("keyExchange", eid, aid, Y_P) to P.

On message ("input", c_{inp}) from P with $pid_P \in \mathcal{P}$:
- Call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{"input"}, P, c_{\text{inp}}))$ and receive (ν, P', out)
- If $P' = \bot$, do nothing.
- Otherwise, call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, \text{"attestOutput"})$ and receive aid, send ("output", aid, ν, out) to P'.

Protocol $\pi_f^{\mathsf{comp}}[H]$ for P with $pid_P \in \mathcal{P}$

On first activation with identifier sid:
- Set $\mathsf{oids} = \emptyset$.
- Draw $x \xleftarrow{\$} \mathbb{Z}_p$, compute $X_P := g^x$ and send ("keyExchange", X_P) to H.
- Await ("keyExchange", eid, aid, Y_P) from H.
- Verify that $\mathcal{G}_{\text{att}}.\texttt{VerifyAttestation}(sid, \mathsf{prog}_f, eid, aid, (X_P, Y_P)) \stackrel{?}{=} 1$. Abort otherwise.
- Compute and store $k = {Y_P}^x$.
- Send ("eid", eid) to each P' for each $pid_{P'} \in \mathcal{P} \setminus \{pid_P\}$.
- For each P' with $pid_{P'} \in \mathcal{P} \setminus \{pid_P\}$, await ("eid", eid') and verify that $eid' = eid$. Abort otherwise.

On input m with identifier sid:
- Compute $c_{\text{inp}} \leftarrow \mathsf{AE.Enc}(k, m)$.

– Send (“input”, c_{inp}) to H.
On message (“output”, aid, ν, out) from H:
– Verify that $\mathcal{G}_{\text{att}}.\texttt{VerifyAttestation}(sid, \mathsf{prog}_f, eid, aid, (\nu, P, \mathsf{out})) \stackrel{?}{=} 1$. Abort otherwise.
– Also verify that $\nu \notin \mathsf{oids}$. Abort otherwise.
– Set $\mathsf{oids} \coloneqq \mathsf{oids} \cup \{\nu\}$.
– Output out.

Here, the single input interface for inputs m is used in the same way as the single input interface inp in $\mathcal{F}_f$. In our protocols and simulators, we omit the session identifiers in protocol messages for better readability. When writing, *e.g.*, (“keyExchange”, X_P), we implicitly assume the message to be (“keyExchange”, sid, X_P).

Note that in the first phase of our protocol, a key exchange is performed between the parties and the enclave. While in principle, any two-round key exchange protocol can be used to achieve the same result, for simplicity, we restrict ourselves to the Diffie-Hellman key exchange.

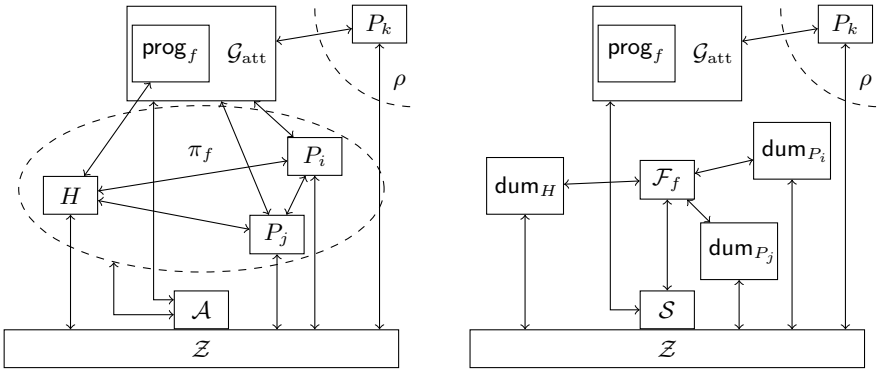
Note that in the protocol and the ideal functionality, we only consider public outputs. While in principle, private outputs can be achieved by encrypting an output to a party using that party's key k_P using an authenticated encryption scheme and replay detection (which also allows for removing the attestation for these outputs), we also chose to omit this for simplicity.

Figure 4 illustrates the interplay between the parties of Constr. 1. While only the host has direct access to its enclave with prog_f installed, all parties can call the interfaces of $\mathcal{G}_{\text{att}}$. The adversary is asked only once by $\mathcal{G}_{\text{att}}$ to provide a MAC and a signature algorithm and there is no communication between them afterwards, except for communication via corrupted parties.

Theorem 5. *Assuming* AE *is an INT-CTXT, IND-CPA secure, perfectly correct encryption scheme and the DDH assumption holds, π_f UC-realizes $\mathcal{F}_f$ in the presence of $\mathcal{G}_{\text{att}}$ under static corruption.*

The idea behind proving Thm. 5 is that all attestations of enclave outputs used in this protocol are tied to a party and can therefore only be verified by this party within this protocol session. As all attestations can only be verified within the current session, the simulator can issue valid attestations to a party, which it can only verify within this session. If a corrupted party does this, the simulator learns the $\mathcal{G}_{\text{att}}.\texttt{VerifyAttestation}$ call and can overwrite the return value to the party. This means that effectively, during simulation, the enclave does not need to be installed with $\mathcal{G}_{\text{att}}$, but the simulator can simulate it for corrupted parties. This contrasts with the model of Pass, Shi, and Tramèr [PST17], where attestations are signatures, that can be verified by any party even from other sessions. We defer a more detailed proof of Thm. 5 to App. E.

Remark 3. We want to emphasize that Constr. 1 only makes use of a single enclave, therefore allowing enclaves to be used as a trusted third party. While



(a) Real world (b) Ideal world

Fig. 4: Illustration of the components relevant in Thm. 5 in a GUC-like setting. While H, P_i, and P_j are representative of protocol participants in π_f, P_k stands for external parties from different protocols (exemplarily called ρ) and sessions. The dummy party of party P is denoted with dum_P.

intuitively, this should be achievable using any enclave model, Pass, Shi, and Tramèr [PST17] have shown that using $\mathcal{G}_{\text{att}}^{\text{PST}}$ at least two enclaves are necessary to realize even simple UC functionalities, such as commitments. Therefore, with Thm. 5, we can circumvent this impossibility result using our enclave model $\mathcal{G}_{\text{att}}$. In essence, Pass *et al.*'s impossibility result is caused by the fact that, with $\mathcal{G}_{\text{att}}^{\text{PST}}$, the simulator has no way of lying about the output of an enclave program. All outputs have an attestation, and any party, including honest parties in different protocols, can verify these attestations. We are able to circumvent this impossibility result by allowing outputs which are not verifiable by every party and instead selectively allow parties to verify an output. This allows the simulator to lie about outputs of an enclave program to corrupted parties while still not allowing it to create attestations for fake messages that might be verifiable by honest parties, as long as the enclave program only ever issues such attestations for parties participating in this protocol.

5 Global Public Verifiability

In this section, we explore the notion of global public verifiability. Intuitively, this means that honest parties can (jointly) evaluate a (probabilistic) function and create an artifact that enables them to convince any party, regardless of whether they participated in the protocol session, that a certain value was the result of the function evaluation. Corrupted parties should not have any other way of creating such an artifact besides essentially also evaluating the function honestly. We use the term "global" to emphasize that any party from any session can perform a verification, and to differentiate from other variants of public verifiability, in which the verifiers are modeled as members of the same protocol session who

remain idle during the protocol execution. We write public verifiability for short whenever we mean global public verifiability.

In this section, we present a protocol that uses $\mathcal{G}_{\text{att}}$ to achieve public verifiability. We also give a formal definition of public verifiability in our setting and present an impossibility result for the model of [PST17].

5.1 General MPC with (Globally) Publicly Verifiable Output

The enclave program prog_f presented in the previous section only uses designated-verifier attestations via the `AddAttestation` interface. If an output is supposed to be *publicly* verifiable, then the `AddAttestationAll` interface provided by $\mathcal{G}_{\text{att}}$ can be used. In this section, we extend the enclave program and real protocol to include a publicly verifiable attestation and prove its security.

However, three challenges arise: Once we consider public verifiability, we also need to provide the security of protocol executions involving only corrupted parties, since the verifier may be honest. In such a case, the parties could run the protocol multiple times (including the use of multiple enclaves), resulting in multiple outputs, until they are satisfied with the result. To capture this in the ideal functionality, we define $\mathcal{F}_{\mathsf{m}f}$, the multi-session variant of $\mathcal{F}_f$, in Def. 3, where any differences to the previous section are highlighted. The idea is like that of $\mathcal{F}_{\text{mcom}}$ by Canetti and Fischlin [CF01]: $\mathcal{F}_{\mathsf{m}f}$ captures multiple "sub-sessions" of $\mathcal{F}_f$ in one ideal functionality with the help of an additional function evaluation ID fid. This fid is unique for every session and is used as an input together with the input value inp as well as in messages to address the correct execution of $\mathcal{F}_f$.

Definition 3. *The Ideal Functionality for Multiple Executions of f.* Let $\mathcal{P}$ be a set of parties. Let t be a polynomial and let $f\colon \{0,1\}^* \times \{0,1\}^{t(\kappa)} \times \mathcal{P} \times \{0,1\}^* \mapsto \{0,1\}^* \times (\mathcal{P} \cup \{\bot\}) \times \{0,1\}^*$ be an efficiently computable function, then we define the ideal functionality $\mathcal{F}_{\mathsf{m}f}$ as follows:

$\mathcal{F}_{\mathsf{m}f}$

- On input (fid, inp) from party P with $pid_P \in \mathcal{P}$,
  - If fid has not been recorded yet, set $\text{state}_{fid} \coloneqq \varepsilon$ and record fid.
  - Notify the adversary about this input by sending $(fid, |\mathsf{inp}|, P)$ and wait for the adversary to acknowledge this.
  - Sample $r \xleftarrow{\$} \{0,1\}^{t(\kappa)}$.
  - Compute $(\text{state}'_{fid}, P', \mathsf{out}) \coloneqq f(\text{state}_{fid}, r, P, \mathsf{inp})$.
  - Set $\text{state}_{fid} \coloneqq \text{state}'_{fid}$.
  - If $P' \neq \bot$, send a public delayed output (fid, out) to P'.
  - If $P' = \bot$, send ("`output`", fid) to $\mathcal{A}$.

The motivation behind $\mathcal{F}_{\mathsf{m}f}$ is as follows: In a real execution where *all* parties are corrupted, they could just execute the real protocol over and over (possibly varying inputs or encountering different randomness used by the enclave in each repetition), thereby possibly biasing the result compared to a single execution

without repetition. In order to be able to still prove security in this setting, we need to admit the simulator the same power. We do so by also repeatedly allowing it to evaluate f multiple times within $\mathcal{F}_{\mathsf{m}f}$. In the case that there is at least one honest party participating in the protocol, this scenario is preventable, as the honest party could refuse to participate in the protocol more than once. $\mathcal{F}_{\mathsf{m}f}$ then essentially offers the same guarantees as $\mathcal{F}_f$.

Secondly, when trying to show that the protocol $\pi^{\mathsf{pub}}_{\mathsf{mult},f}$ with publicly verifiable outputs realizes $\mathcal{F}_{\mathsf{m}f}$ in the presence of $\mathcal{G}_{\mathrm{att}}$, the outputs computed by $\mathcal{F}_{\mathsf{m}f}$ and $\mathcal{G}_{\mathrm{att}}$ might differ. In particular, this can happen with any probabilistic functionality, *e.g.*, a coin-toss. While $\mathcal{F}_{\mathsf{m}f}$ might randomly sample a bit $b = 0$, the enclave could have sampled $b = 1$ at the same time, resulting in an inconsistency.

This gap cannot be bridged by the simulator for several reasons: First, the simulator is an adversarial entity. When considering public verifiability, the simulator as ideal-world adversary must not be able to (arbitrarily) influence outputs that may be verified by honest external parties, as the effect of its simulation crosses protocol boundaries and may affect the security guarantees given to honest parties not participating in the protocol. Leaving this aside, we observe that any power given to the simulator is also available to the real-world adversary in a hybrid model or the environment in other sessions of protocols that use $\mathcal{G}_{\mathrm{att}}$ (through the adversary), leading to an unsound notion.

We thus delegate output reconciliation of $\mathcal{F}_{\mathsf{m}f}$ and $\mathcal{G}_{\mathrm{att}}$ to $\mathcal{F}_{\mathsf{m}f}$, or more precisely, to a *shelled* variant $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ of $\mathcal{F}_{\mathsf{m}f}$. To this end, the `AddAttestationAll` interface gives special treatment to $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$, allowing it to choose output values of an enclave. In the simulation without public verifiability, the simulator can create all enclave messages, and no enclave needs to be installed. Therefore, this interface is only necessary when publicly verifiable attestations are involved. Let us elaborate the implications for simulation-based security. First, the simulator is still required to extract the inputs of corrupted parties and to provide them to $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$. Only then, $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ will use the interface provided by $\mathcal{G}_{\mathrm{att}}$ to adjust outputs. Taking a closer look, by the definition of $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$, the interface is restricted to settings where only certain enclave programs are installed—namely those that perform the actual computation *exactly* like $\mathcal{F}_{\mathsf{m}f}$ (for the same f). Consequently, an enclave provided with the same inputs of the involved parties (corrupted and honest) using the same randomness as $\mathcal{F}_{\mathsf{m}f}$ would produce *exactly* the same (publicly verifiable) outputs. Apart from internally evaluating f on the inputs, the only other input-dependent computations are the generation and transmission of the ciphertexts containing the (honest) parties' inputs as well as the ciphertexts for the output. For honest parties, these are still simulated by the simulator. Thus, the security proof ensures that no unintentional information is leaked at this stage (apart from the inputs' length), and the only information flow from the ideal functionality are the computation results of $\mathcal{F}_{\mathsf{m}f}$. Consequently, meaningful security guarantees are provided. For parties participating in the protocol execution, the *same* guarantees as in a standard execution hold, because they get their outputs via $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ unmodified and do not rely on the publicly verifiable attestation at all.

We also need to take care of certain, mainly syntactical, details. In particular, the ideal functionality must come up with an enclave ID and input it into $\mathcal{G}_{\text{att}}$ when calling `AddAttestationAll`. This way, $\mathcal{G}_{\text{att}}$ can answer verification requests for this enclave ID without the enclave having to be installed.

The third challenge is the possibility of an impersonation attack in the real execution. Since only the enclave host has a direct connection to their enclave and can give inputs to it, a corrupted host might impersonate the other protocol participants, give inputs in their name and not involve them in the protocol execution at all. In a non-publicly verifiable setting, the honest parties that were left out of the protocol execution could not be convinced of any attestation related to this execution, allowing them to halt or abort. In the publicly verifiable case, however, external verifiers could be convinced that this protocol execution took place and that all parties that were supposed to be involved actually participated. To prevent this attack, a global registry functionality $\mathcal{G}_{\text{reg}}$ (Def. 5) is used as a PKI where parties can register their verification keys. Only when all honest parties have registered their keys, the enclave is installed via the host. This allows an appropriate enclave program parameterized with the parties' public keys to identify the actual participation of parties through (encrypted) fresh signatures.

To implement these changes necessary compared to the previous section, we introduce an ideal shell $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ around $\mathcal{F}_{\mathsf{m}f}$ in Def. 4. The shell uses the global functionality $\mathcal{G}_{\text{reg}}$ defined in Def. 5 as a PKI. $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ intercepts the first invocation of $\mathcal{F}_{\mathsf{m}f}$ to register parties at $\mathcal{G}_{\text{reg}}$. Additionally, it calls the `AddAttestationAll` interface of $\mathcal{G}_{\text{att}}$ for the publicly verifiable outputs of $\mathcal{F}_{\mathsf{m}f}$. Since corrupted parties might install multiple enclaves and have them evaluate a function before deciding which result to use, $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ lets the adversary choose one from a selection of randomly drawn *eid*s to be used for the attestation.

Definition 4. *The Ideal Shell.* Let $\mathcal{P}$ be a set of parties. Let $\mathcal{F}_{\mathsf{m}f}$ be the ideal functionality defined in Def. 3. We define the ideal shell $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ as follows:

$\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$

- On input (fid, inp) from honest party P, if $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ has not registered a public key for all parties yet:
  - Send ("`setup`", fid, pid_P) to the adversary.
  - For each unregistered party P':
    - Wait for a response ("`register`", $fid, pid_{P'}$) from the adversary.
    - Generate $(vk_{P'}, \cdot) \leftarrow \Sigma.\mathsf{Gen}(1^\kappa)$ and call $\mathcal{G}_{\text{reg}}.\texttt{Register}(vk_{P'}, P'_{\mathsf{pub}})$, where P'_{pub} is the extended identity of P', except that the code is replaced with $\pi^{\mathsf{pub}}_{\mathsf{mult},f}$.
    - Send ("`registered`", $fid, pid_{P'}$) to the adversary.
  - Wait for a backdoor message ("`continue`", fid, pid_P) from the adversary after $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ has already sent a backdoor message ("`eid`", $fid, \cdot$) to the adversary in a different activation.
  - Forward the input (fid, inp) to $\mathcal{F}_{\mathsf{m}f}$.
- On backdoor message ("`candidateEid`") from the adversary:

- For each corrupted party P with $pid_P \in \mathcal{P}$, call $\mathcal{G}_{\text{reg}}.\texttt{Retrieve}(P_{\mathsf{pub}})$ and receive ("`retrieve`", P_{pub}, vk_P) from $\mathcal{G}_{\text{reg}}$.
- If any $vk_P = \bot$ or if $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$ has not registered the public keys for all honest parties yet, ignore this backdoor message.
- Draw $eid' \xleftarrow{\$} \{0,1\}^\kappa$.
- Send ("`candidateEid`", eid') to the adversary.

– On backdoor message ("`chooseEid`", fid, eid') from the adversary:
- If there already was a backdoor message ("`eid`", $\cdot$, eid') or ("`eid`", fid, $\cdot$) to the adversary or if there was no previous backdoor message ("`candidateEid`", eid') to the adversary, ignore this backdoor message.
- Set $eid_{fid} := eid'$ and send ("`eid`", fid, eid_{fid}) to the adversary.

– On public delayed output (fid, out) from $\mathcal{F}_{\mathsf{mf}}$ to some party P with $pid_P \in \mathcal{P}$:
- Draw $\nu \xleftarrow{\$} \{0,1\}^\kappa$.
- $aid \leftarrow \mathcal{G}_{\text{att}}.\texttt{AddAttestationAll}(eid_{fid}, (\nu, P_{\mathsf{pub}}, \mathsf{out}), \mathsf{prog}_f[\{(P_{\mathsf{pub}}, vk_P)\}_{pid_P \in \mathcal{P}}])$.
- Change the value of the public delayed output to $(fid, eid_{fid}, \mathsf{prog}_f[\{(P_{\mathsf{pub}}, vk_P)\}_{pid_P \in \mathcal{P}}], aid, \nu, \mathsf{out})$.[1]

– Forward all other inputs, outputs and backdoor messages to and from $\mathcal{F}_{\mathsf{mf}}$ unmodified.

Definition 5. *PKI Functionality* $\mathcal{G}_{\text{reg}}$.

$\mathcal{G}_{\text{reg}}$

On input $\texttt{Register}(v)$ from an ITI with extended identity P:
– If a record (P, v) is already stored, ignore call.
– Notify the adversary about this input by sending it ("`registered`", P) and wait for the adversary to acknowledge this.
– Store (P, v).
– Ignore all further $\texttt{Register}(\cdot)$ calls from ITIs with extended identity P.

On input $\texttt{Register}(v, P)$ from an ITI with code $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$:
– If a record (P, v) is already stored, ignore call.
– Notify the adversary about this input by sending it ("`registered`", P) and wait for the adversary to acknowledge this.
– Store (P, v).

[1] Specifically, $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$ forwards the backdoor messages and the output that are part of $\mathcal{F}_{\mathsf{mf}}$'s delayed output while translating back and forth between $\mathcal{F}_{\mathsf{mf}}$'s output value (fid, out) and the output value $(fid, eid, \mathsf{prog}, aid, \nu, \mathsf{out})$ that is visible to the outside. For our definition of public verifiability, it is important to note that $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$ therefore produces output if and only if $\mathcal{F}_{\mathsf{mf}}$ produces output.

On input `Retrieve`(P) from any ITI:
- If a record (P, v) is stored, respond with a public delayed output ("`retrieve`", P, v). Otherwise, respond with a public delayed output ("`retrieve`", $P, \perp$).

$\mathcal{G}_{\text{reg}}$ is a modification of Canetti, Shahaf and Vald's certificate authority functionality [CSV16, Figure 3]. Note that, while Canetti, Shahaf and Vald's functionality stores at most one value per PID, the $\mathcal{G}_{\text{reg}}$ functionality that we use stores up to one value per extended identity. Another difference is that we included an interface for the functionality $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ introduced in Def. 4 to register under different identities. The shell uses the interface to mask the code of the registering ITI as the code of the respective real protocol. Without this interface, the environment could see that the code of the registered ITIs is different between the real and the ideal world and could thus distinguish them. On the other hand, because this interface can only be used by $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$, the environment cannot abuse the interface via maliciously designed ideal functionalities to impersonate parties.

The enclave program and the protocols for the enclave host and the other parties are shown in Constr. 2. The structure is similar to that of Constr. 1 and differences are again highlighted. The most notable additions are the introduction of the PKI via $\mathcal{G}_{\text{reg}}$ and the replacement of $\mathcal{G}_{\text{att}}$.`VerifyAttestation` with $\mathcal{G}_{\text{att}}$.`VerifyAttestationAll` for the attestation of the output. The attestation of the key exchange remains designated-verifier. Even if the attestation interface for publicly verifiable attestations is now used for the output, we still rely on designated-verifier attestations in the key exchange to circumvent the impossibility result by Pass *et al.* addressed in Sect. 4 and there is still only one enclave necessary in Constr. 2. The enclave program is now parameterized with the extended identities and public keys of the protocol parties.

Please note that $\mathcal{F}_{\mathsf{m}f}$ can be realized without $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ with enclaves (see Sect. 4) for designated verifiers (without public verifiability).

Construction 2 *The protocol* $\pi^{\mathsf{pub}}_{\mathsf{mult},f}$. Let $\mathcal{P}$ be a set of parties and let H be a party with an enclave. ($pid_H \in \mathcal{P}$ is not necessary. If $pid_H \notin \mathcal{P}$, we are in the case, where a third party has the enclave.) Let t be a polynomial and let $f\colon \{0,1\}^* \times \{0,1\}^{t(\kappa)} \times \mathcal{P} \times \{0,1\}^* \mapsto \{0,1\}^* \times (\mathcal{P} \cup \{\perp\}) \times \{0,1\}^*$ be an efficiently computable function. We define two parts $\pi^{\mathsf{pub,comp}}_{\mathsf{mult},f}$ and $\pi^{\mathsf{pub,host}}_{\mathsf{mult},f}$ of the protocol $\pi^{\mathsf{pub}}_{\mathsf{mult},f}$, as well as an enclave program prog_f, that is used in the protocol. $\pi^{\mathsf{pub,comp}}_{\mathsf{mult},f}$ is executed by all parties in $\mathcal{P}$, while $\pi^{\mathsf{pub,host}}_{\mathsf{mult},f}$ is executed by the party hosting the enclave. If $pid_H \in \mathcal{P}$, H executes both protocols simultaneously, potentially sending messages to itself from $\pi^{\mathsf{pub,comp}}_{\mathsf{mult},f}$ to $\pi^{\mathsf{pub,host}}_{\mathsf{mult},f}$ or vice versa.

Enclave program $\mathsf{prog}_f[\{(P, vk_P)\}_{pid_P \in \mathcal{P}}]$
Initialize state $= \varepsilon, \mathsf{L} = \emptyset$

On input ("keyExchange", P, X_P): sample $y_P \xleftarrow{\$} \mathbb{Z}_p$ and $s_P \xleftarrow{\$} \{0,1\}^\kappa$, store X_P and $k_P = X_P^{y_P}$, and output (P, X_P, g^{y_P}, s_P).
On input ("attest", P): If k_P, y_P and s_P have been previously stored, compute $aid_P \leftarrow \texttt{AddAttestation}(P, (P, X_P, g^{y_P}, s_P))$, output aid_P.
On input ("identify", P, c_{ident}): Compute $\sigma := \mathsf{AE.Dec}(k_P, c_{\text{ident}})$. If $\Sigma.\mathsf{Verify}(vk_P, s_P, \sigma) = 1$, mark P as identified and output 1. Otherwise, output 0.
On input ("input", P, c_{inp}): Verify $c_{\text{inp}} \notin \mathsf{L}$, abort otherwise. Set $\mathsf{L} := \mathsf{L} \cup \{c_{\text{inp}}\}$. Compute $\mathsf{inp} := \mathsf{AE.Dec}(k_P, c_{\text{inp}})$. Ignore input if $\mathsf{inp} = \bot$ or if not all parties are identified. Draw $r \xleftarrow{\$} \{0,1\}^{t(\kappa)}$. Compute $(\text{state}', P', \mathsf{out}) := f(\text{state}, r, P, \mathsf{inp})$. Store state := state′. Draw $\nu \xleftarrow{\$} \{0,1\}^\kappa$, output (ν, P', out) and store it as the latest output.
On input ("attestOutput"): For the latest output (ν, P', out), verify that $P' \neq \bot$. Then, compute $aid \leftarrow \texttt{AddAttestationAll}(\nu, P', \mathsf{out})$, output aid and set the latest stored output to $\bot$.

Protocol $\pi^{\mathsf{pub,host}}_{\mathsf{mult},f}$ for the enclave host H

On first activation with identifier sid, fid:
- For each party P with $pid_P \in \mathcal{P}$:
  - Await ("keyExchange", fid, X_P) from each party P.
  - Call $\mathcal{G}_{\text{reg}}.\texttt{Retrieve}(P)$ and receive ("retrieve", P, vk_P) from $\mathcal{G}_{\text{reg}}$.
  - Abort if $vk_P = \bot$.
- H calls $\mathcal{G}_{\text{att}}.\texttt{Install}(sid, \mathsf{prog}_f[\{(P, vk_P)\}_{pid_P \in \mathcal{P}}])$ and receives and stores eid.
- For each party P with $pid_P \in \mathcal{P}$:
  - Call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{"keyExchange"}, P, X_P))$ and receive (P, X_P, Y_P, s_P) from $\mathcal{G}_{\text{att}}$.
  - Call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{"attest"}, P))$ and receive aid from $\mathcal{G}_{\text{att}}$.
  - Send ("keyExchange", fid, eid, aid, Y_P, s_P) to P and receive ("identify", fid, c_{ident}) from P.
  - Call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{"identify"}, P, c_{\text{ident}}))$.
  - Abort if the output from $\mathcal{G}_{\text{att}}$ is 0.

On message ("input", fid, c_{inp}) from P with $pid_P \in \mathcal{P}$:
- Call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{"input"}, P, c_{\text{inp}}))$ and receive $(fid, \nu, P', \mathsf{out})$
- If $P' = \bot$, do nothing.
- Otherwise, call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, \text{"attestOutput"})$ and receive aid. Send ("output", $fid, aid, \nu, \mathsf{out}$) to P'.

Protocol $\pi^{\mathsf{pub,comp}}_{\mathsf{mult},f}[H]$ for P with $pid_P \in \mathcal{P}$:

On first activation with identifier sid, fid:

- Set $\mathsf{oids} = \emptyset$
- Draw $(vk_P, sk_P) \leftarrow \Sigma.\mathsf{Gen}(1^\kappa)$ and call $\mathcal{G}_{\text{reg}}.\texttt{Register}(vk_P)$.
- Draw $x \xleftarrow{\$} \mathbb{Z}_p$, compute $X_P := g^x$ and send ("keyExchange", fid, X_P) to H.
- Await ("keyExchange", fid, eid, aid, Y_P, s) from H.
- For each party P' with $pid_{P'} \in \mathcal{P} \setminus \{pid_P\}$, call $\mathcal{G}_{\text{reg}}.\texttt{Retrieve}(P')$ and receive ("retrieve", P', $vk_{P'}$) from $\mathcal{G}_{\text{reg}}$.
- Abort if any $vk_{P'} = \bot$.
- Verify that $\mathcal{G}_{\text{att}}.\texttt{VerifyAttestation}(sid, \mathsf{prog}_f[\{(P', vk_{P'})\}_{pid_{P'} \in \mathcal{P}}], eid, aid, (P, X_P, Y_P, s)) \stackrel{?}{=} 1$. Abort otherwise.
- Compute and store $k := Y_P^x$.
- Compute $\sigma \leftarrow \Sigma.\mathsf{Sign}(sk_P, s)$ and $c_{\text{ident}} \leftarrow \mathsf{AE.Enc}(k, \sigma)$ and send ("identify", fid, c_{ident}) to H.
- Send ("eid", fid, eid) to each P' for each $pid_{P'} \in \mathcal{P} \setminus \{pid_P\}$.
- For each P' with $pid_{P'} \in \mathcal{P} \setminus \{pid_P\}$, await ("eid", fid, eid') and verify that $eid' = eid$. Abort otherwise.

On input m with identifier sid, fid:

- Compute $c_{\text{inp}} \leftarrow \mathsf{AE.Enc}(k, m)$.
- Send ("input", fid, c_{inp}) to H.

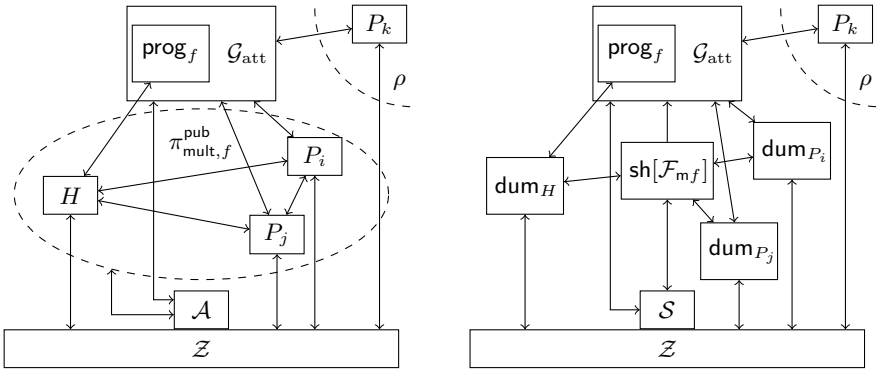
On message ("output", fid, aid, ν, out) from H:

- Verify that $\mathcal{G}_{\text{att}}.\texttt{VerifyAttestationAll}(sid, \mathsf{prog}_f[\{(P', vk_{P'})\}_{pid_{P'} \in \mathcal{P}}], eid, aid, (\nu, P, \mathsf{out})) \stackrel{?}{=} 1$. Abort otherwise.
- Also verify that $\nu \notin \mathsf{oids}$. Abort otherwise.
- Set $\mathsf{oids} := \mathsf{oids} \cup \{\nu\}$.
- Output (fid, eid, $\mathsf{prog}_f[\{(P, vk_P)\}_{pid_P \in \mathcal{P}}]$, aid, ν, out).

The parties of Constr. 2 are illustrated in Fig. 5. Compared with Fig. 4, the ideal functionality is replaced with $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$, which can overwrite outputs via the $\mathcal{G}_{\text{att}}.\texttt{AddAttestationAll}$ interface. Any party can call $\mathcal{G}_{\text{att}}$ to verify a publicly verifiable attestation.

Theorem 6. *Let ξ be an identity bound that excludes the extended identity of an instance of $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ with the test SID. Assuming AE is an INT-CTXT, IND-CPA secure, perfectly correct encryption scheme and the DDH assumption holds, the protocol $\pi^{\mathsf{pub}}_{\mathsf{mult},f}$ ξ-UC-realizes $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ in the presence of $\mathcal{G}_{\text{att}}$ and $\mathcal{G}_{\text{reg}}$ under static corruption.*

To prove Thm. 6, the idea is to not actually install enclaves for honest hosts or corrupted hosts that install a program of the correct form, but to let the simulator create designated-verifier attestations in its head. However, publicly verifiable attestations must be verifiable at $\mathcal{G}_{\text{att}}$. To make this possible, $\mathcal{G}_{\text{att}}.\texttt{AddAttestationAll}$ is called in the ideal shell $\mathsf{sh}[\cdot]$, which adds a valid attestation to an output, even if the corresponding enclave has never been installed in the simulation. This is a



(a) Real world (b) Ideal world

Fig. 5: Illustration of the components relevant in Thm. 6 in a GUC-like setting. The dummy party of party P is denoted with dum_P.

crucial difference to $\mathcal{G}_{\text{att}}^{\text{PST}}$, where no such interface for ideal functionalities with an overwrite mechanism exists. To coordinate the selection of the enclave ID with the ideal functionality, the simulator exchanges backdoor messages with $\mathsf{sh}[\cdot]$. We defer the full proof of Thm. 6 to App. F.

To simplify the presentation of publicly verifiable outputs with the help of $\mathcal{G}_{\text{att}}$ in this paper, all outputs of $\mathcal{F}_{\text{m}f}$ are publicly verifiable. This contrasts with $\mathcal{F}_f$ (Sect. 4), where the outputs are only verifiable by designated verifiers. A more fine-grained distinction between publicly verifiable outputs and outputs for designated verifiers in one ideal functionality can be achieved by adding an additional parameter to outputs. Depending on the type, either $\mathcal{G}_{\text{att}}$.`AddAttestation` or $\mathcal{G}_{\text{att}}$.`AddAttestationAll` must be called.

5.2 Definition of Global Public Verifiability

So far, the fact that the output of the computation was publicly verifiable was only implicit through the call of $\mathcal{G}_{\text{att}}$.`AddAttestationAll` in the shell $\mathsf{sh}[\mathcal{F}_{\text{m}f}]$. In this section, we give a more explicit definition of public verifiability and show that our model of attested execution fulfills this definition. In App. H, we give a formal proof to show that $\mathcal{G}_{\text{att}}^{\text{PST}}$ does not provide public verifiability.

Our definition of public verifiability is loosely based on [BDO14, Def. 1 (Auditable Correctness)]. We formalize the idea that any external party should be able to verify if the result of a protocol π was correctly computed. For the sake of making the definition comprehensible, we limit the scope in this work to protocols that specifically use $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{att}}^{\text{MAC}}$ or $\mathcal{G}_{\text{att}}$ in combination with $\mathcal{G}_{\text{reg}}$ to achieve public verifiability.

To define which output is the correct one, we use the ideal functionality $\mathcal{F}$ that π is designed to UC-realize. The advantage of using $\mathcal{F}$ is that its outputs are well-defined, regardless of whether the party receiving the output is corrupted.

To pass the necessary inputs to $\mathcal{F}$ to compute the outputs, we use the fact that, for UC-emulation, there must be a simulator $\mathcal{S}$ that can extract inputs from corrupted parties. Thus, with the combination of $\mathcal{F}$ and $\mathcal{S}$, we get an unambiguous set of target values Y that any verifying party should be able to recognize as the correct outputs, and the verifying party should recognize all other values as incorrect. In the case that $\mathcal{F}$ did not generate output, all values are considered incorrect. Due to this way of defining Y, our definition also extends to the case that all parties in the protocol session are corrupted. Note that $\mathcal{S}$ does not have to report the outputs of $\mathcal{F}$ to the environment for Y to be well-defined.

In our definition, $\mathcal{F}$ is explicitly allowed to be probabilistic. One issue with probabilistic functionalities is that in the case where all parties are corrupted, the protocol could be executed multiple times until the adversary is satisfied with the result. This takes away from the publicly verifiable character of the output. However, we still include this case in our definition, since, when an enclave chooses the randomness in every protocol instance, some entropy remains. This shows that, with enclaves, we can achieve more than what is defined as auditable correctness in [BDO14], since Baum *et al.* cannot give any guarantees for probabilistic functionalities if all parties are corrupted.

We model the verifying parties via a PPT algorithm V that receives some information about the execution with functionality $\mathcal{F}$ after the execution ended and then, based on this information, determines whether some supposed output y_i' was an actual output from the set Y. The information is split into a trusted part t and an untrusted part U. The idea is that V should still reject an incorrect output, even if it receives adversarially chosen output y_i' and untrusted information U'. The trusted information includes the session ID, the public key of the $\mathcal{G}_{\text{att}}^*$ variant if $\mathcal{G}_{\text{att}}^*$ has one, and the mapping between identities and public keys registered at the PKI. We assume that the verifying parties are able to obtain this information without any manipulation from an adversary. The outputs of the enclaves involved in the session do not have to be trusted, because the enclave outputs have signatures that can be verified with the public key of $\mathcal{G}_{\text{att}}^*$, or they have attestation IDs that can be verified directly at $\mathcal{G}_{\text{att}}^*$.

Definition 6. *Special Enclave-Based Global Public Verifiability.* Let ξ be an identity bound that excludes all extended identities with code $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$. Let $\mathcal{F}$ be an ideal functionality, such that the output values produced by $\mathcal{F}$ only depend on the inputs to $\mathcal{F}$ and the contents of $\mathcal{F}$'s random tape. For $\mathcal{G}_{\text{att}}^* \in \{\mathcal{G}_{\text{att}}^{\text{PST}}, \mathcal{G}_{\text{att}}^{\text{MAC}}, \mathcal{G}_{\text{att}}\}$, let γ^* denote the combination of the ideal protocols for $\mathcal{G}_{\text{att}}^*$ and $\mathcal{G}_{\text{reg}}$.[2] We say that a protocol π is specially publicly verifiable with respect to $\mathcal{F}$ using $\mathcal{G}_{\text{att}}^*$ and $\mathcal{G}_{\text{reg}}$ if

- there exists a protocol $\phi \in \{\text{IDEAL}(\mathcal{F}), \text{IDEAL}(\mathsf{sh}[\mathcal{F}])\}$ and a PPT verifier V such that
- for every PPT adversary $\mathcal{A}$, there is a PPT simulator $\mathcal{S}$ such that for every PPT ξ-identity-bounded environment $\mathcal{Z}$, it holds that the ensembles

[2] By "combination" we mean that γ^* provides two interfaces that are functionally equivalent to the interfaces of the ideal protocols for $\mathcal{G}_{\text{att}}^*$ and $\mathcal{G}_{\text{reg}}$.

$\mathsf{Exec}(\mathsf{M}[\pi, \gamma^*], \mathcal{A}, \mathcal{Z})$ and $\mathsf{Exec}(\mathsf{M}[\phi, \gamma^*], \mathcal{S}, \mathcal{Z})$[3] are computationally indistinguishable and

– for every PPT adversary $\mathcal{B}$, for every $a \in \{0,1\}^{\leq \mathsf{poly}(\kappa)}$ and for uniformly sampled randomness r, the following holds at the end of $\mathsf{Exec}(\mathsf{M}[\phi, \gamma^*], \mathcal{S}, \mathcal{Z})(\kappa, a; r)$:
  - $\forall(\cdot, y_i) \in Y : \Pr[V^{\mathcal{O}}(1^\kappa, t, U, y_i) = 1] \geq 1 - \mathsf{negl}(\kappa)$,
  - $\Pr[V^{\mathcal{O}}(1^\kappa, t, U', y_i') = 0 \mid (U', y_i') \leftarrow \mathcal{B}(1^\kappa, t, U, Y, \mathrm{view}_{\mathcal{Z}}) \wedge (\cdot, y_i') \notin Y] \geq 1 - \mathsf{negl}(\kappa)$,

  where
  - $\mathcal{O}$ is an oracle that runs $\mathcal{G}_{\mathrm{att}}^*$.`VerifyAttestationAll`, using the information stored inside $\mathcal{G}_{\mathrm{att}}^*$, if $\mathcal{G}_{\mathrm{att}}^* = \mathcal{G}_{\mathrm{att}}$. Otherwise, it does nothing.
  - t is a tuple containing the SID of ϕ, the attestation verification key vk if $\mathcal{G}_{\mathrm{att}}^*$ has one, and all pairs of extended identities and public keys stored in $\mathcal{G}_{\mathrm{reg}}$ that belong to the session with this SID.
  - U is the set of all those tuples of enclave outputs, enclave IDs, enclave programs and signatures or attestation IDs which are stored in $\mathcal{G}_{\mathrm{att}}^*$ and also contained in an output of ϕ.
  - Y is the set of outputs of $\mathcal{F}$.

Since Def. 6 asks for $\mathsf{Exec}(\mathsf{M}[\pi, \gamma^*], \mathcal{A}, \mathcal{Z})$ and $\mathsf{Exec}(\mathsf{M}[\phi, \gamma^*], \mathcal{S}, \mathcal{Z})$ to be computationally indistinguishable, and because the environment is able to retrieve all values that are part of the public verifier's view, the guarantees given for outputs of $\mathcal{F}$ can be applied to outputs of π as well. That means that the public verifier can tell whether a value y_i was part of an actual output of a session of π in the presence of $\mathcal{G}_{\mathrm{att}}^*$ and $\mathcal{G}_{\mathrm{reg}}$. If the outputting party is corrupted but still followed the protocol, the verifier determines if y_i is an output that an honest party would have produced in the same situation.

In Def. 6, the protocol ϕ is limited to either IDEAL($\mathcal{F}$) or IDEAL($\mathsf{sh}[\mathcal{F}]$). We therefore call this notion *special* enclave-based global public verifiability. In App. H, we propose a corresponding *weak* variant of the definition. After formally defining public verifiability, we show that with our model of $\mathcal{G}_{\mathrm{att}}$, we can indeed achieve public verifiability.

Theorem 7. *Assuming* AE *is an INT-CTXT, IND-CPA secure, perfectly correct encryption scheme and the DDH assumption holds, and that Σ is an EUF-CMA secure signature scheme, the protocol $\pi_{\mathsf{mult},f}^{\mathsf{pub}}$ is specially publicly verifiable with respect to $\mathcal{F}_{\mathsf{m}f}$ using $\mathcal{G}_{\mathrm{att}}$ and $\mathcal{G}_{\mathrm{reg}}$.*

This theorem for the most part hinges on the fact that $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ interacts with $\mathcal{G}_{\mathrm{att}}$ directly and is therefore able to create attestations for the correct output values. No other entity is able to create convincing attestations for that session. We defer the details of the proof to App. G.

In contrast, $\mathcal{G}_{\mathrm{att}}^{\mathrm{PST}}$ has no interface for interaction with other ideal functionalities. This means that a session of corrupted parties is able to convince a public verifier by imitating the ideal protocol and the simulator. For certain

[3] $\mathsf{M}[\cdot, \cdot]$ is the protocol from [Bad+20] that models a local protocol with global setup.

functionalities, the level of freedom the corrupted parties have in generating the publicly verifiable output values is too far from the prescribed behavior of the functionality, which breaks the public verifiability property. We show in App. H that this is the case even if we relax the constraint on the ideal protocol ϕ from being one of two specific protocols to a broad category of protocols, and also if we allow protocols to use the setup functionality $\mathcal{G}_{\text{acrs}}$ that Pass *et al.* used to achieve single enclave MPC [PST17, Sect. 2.5].

6 Instantiation of $\mathcal{G}_{\text{att}}$ with Digital Signatures and MACs

Models like $\mathcal{G}_{\text{att}}^{\text{PST}}$ or real-world secure processors like Intel SGX make use of cryptographic primitives like digital signatures to implement remote attestation. To match this approach, in this section, we present an instantiation of our enclave model from Sect. 3 using MACs for designated-verifier attestations and digital signatures for publicly verifiable attestations. In other words, `AddAttestation` resp. `AddAttestationAll` calculates a MAC resp. a signature over the index, the installed program, the enclave identifier, a random attestation identifier and the output using a MAC key resp. a signature key. The MAC key is only known to one remote party and the enclave. To obtain a key, $\mathcal{G}_{\text{att}}^{\text{MAC}}$ has a `Register` interface, in which a key exchange is performed with the remote party. The signature key is only known to the enclave, whereas the verification key can be obtained by any party. Hence, $\mathcal{G}_{\text{att}}^{\text{MAC}}$ lacks interfaces for verifying attestations, as these operations can be performed by parties locally if they hold the correct key.

In addition to presenting $\mathcal{G}_{\text{att}}^{\text{MAC}}$ (Fig. 6), we prove that $\mathcal{G}_{\text{att}}^{\text{MAC}}$ emulates $\mathcal{G}_{\text{att}}$. To prove this, we require a *wrapper* $\mathcal{W}[\cdot]$, (Fig. 7), which accounts for the interface differences between $\mathcal{G}_{\text{att}}^{\text{MAC}}$ and $\mathcal{G}_{\text{att}}$, similar to wrappers used in, *e.g.*, [MK25]. A wrapper is a protocol that is called by parties instead of calling an ideal functionality $\mathcal{F}$ directly. Internally, the wrapper then calls $\mathcal{F}$ in the name of the party after processing the call.[4] In a security proof, the wrapper adds or removes interfaces to match the interfaces of $\mathcal{F}$. We present such a security proof, in which we show the indistinguishability between $\mathcal{G}_{\text{att}}$ and the wrapped $\mathcal{G}_{\text{att}}^{\text{MAC}}$, independent of the protocol and enclave program. Similar to the wrapper, which compensates for the different interfaces, we introduce a *shell* W (Fig. 8) which interacts with the wrapper and the enclave program. We require a shell to initiate the registration process before adding an attestation. The shell is installed by the wrapper instead of the enclave program running in the ideal world. Intuitively, the shell can intercept specific calls from the enclave program to $\mathcal{G}_{\text{att}}^{\text{MAC}}$ to replace it with other instructions. Any output the enclave program would return, the shell can intercept and replace as well. The proposed functionality $\mathcal{G}_{\text{att}}^{\text{MAC}}$ can be used on its own, *i.e.*, without wrapper and shell, in a similar way as $\mathcal{G}_{\text{att}}$.

[4] In particular, the wrapper uses the party's extended identity P_i to call $\mathcal{F}$.

$\mathcal{G}_{\text{att}}^{\text{MAC}}[\Pi, \Sigma]$

On initialize: $T = \emptyset, O = \emptyset, V = \emptyset, \mathsf{key} = \emptyset, (vk, sk) \leftarrow \Sigma.\mathsf{KeyGen}$.

On receive `Getvk`() from some P_j: send vk to P_j.

On input `Install`(idx, prog) from some P_i:
- If P_i is honest, verify that $idx = sid$. Abort otherwise.
- Generate nonce $eid \xleftarrow{\$} \{0,1\}^\lambda$, store $T[eid] := (idx, \mathsf{prog}, \bot, P_i)$, send eid to P_i.

On input `Resume`(eid, inp) from P_i:
- Let $(\cdot, \mathsf{prog}, \mathsf{mem}, P) := T[eid]$ or ignore this input if not found or if $P_i \neq P$.
- Let $(\mathsf{out}, \mathsf{mem}') := \mathsf{prog}(\mathsf{mem}, \mathsf{inp})$ and store $O[eid] := O[eid] \cup \{\mathsf{out}\}$.
- Update $T[eid] := (\cdot, \mathsf{prog}, \mathsf{mem}', P_i)$ and send out to P_i.

On receive `Register`(eid, P_j, m) from P_i:
- If $\mathsf{key}[P_j, eid] \neq \bot$, ignore this call.
- Sample $b \xleftarrow{\$} \mathbb{Z}_p$ and store $\mathsf{key}[P_j, eid] = m^b$.
- If P_j is corrupted, send m^b to the adversary.
- Return $(g^b, \sigma \leftarrow \Sigma.\mathsf{Sign}(sk, (\text{“}\texttt{register}\text{”}, P_j, eid, (m, g^b))))$.

Enclave-Internal Operations

On receive `AddAttestation`(P, out) from a TEE with EID eid and program prog:
- Verify that $\mathsf{key}[P, eid] \neq \bot$, ignore call otherwise.
- Verify that $\mathsf{out} \in O[eid]$, ignore call otherwise.
- Let $(idx, \mathsf{prog}, \cdot, P_i) := T[eid]$.
- If $(P, idx, \mathsf{prog}, eid, \cdot, \mathsf{out})$ is not stored in V, sample $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$ and generate $aid_2 := \Pi.\mathsf{Mac}(\mathsf{key}[P, eid], (\text{“}\texttt{attest}\text{”}, idx, \mathsf{prog}, eid, aid_1, \mathsf{out}))$.
- Set $V := V \cup \{(P, idx, \mathsf{prog}, eid, aid := aid_1 || aid_2, \mathsf{out})\}$.
- Otherwise, get aid from V.
- Return aid.

On receive `AddAttestationAll`(out) from a TEE with EID eid and program prog:
- Verify that $\mathsf{out} \in O[eid]$, ignore call otherwise.
- Let $(idx, \mathsf{prog}, \cdot, P_i) := T[eid]$.
- If $(idx, \mathsf{prog}, eid, \cdot, \mathsf{out})$ is not stored in V, sample $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$ and generate $aid_2 := \Sigma.\mathsf{Sign}(sk, (\text{“}\texttt{attest}\text{”}, idx, \mathsf{prog}, eid, aid_1, \mathsf{out}))$.
- Set $V := V \cup \{(idx, \mathsf{prog}, eid, aid := aid_1 || aid_2, \mathsf{out})\}$.
- Otherwise, get aid from V.
- Return aid.

Ideal Functionality Operations

On receive `AddAttestationAll`($eid, \mathsf{out}, \mathsf{prog}$) from an ideal functionality $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$:
- Let idx be the sid of $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$.
- Sample $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$ and generate $aid_2 := \Sigma.\mathsf{Sign}(sk, (\text{“}\texttt{attest}\text{”}, idx, \mathsf{prog}, eid, aid_1, \mathsf{out}))$.
- Set $V := V \cup \{(idx, \mathsf{prog}, eid, aid := aid_1 || aid_2, \mathsf{out})\}$. // Always generate a record.
- Return aid.

Fig. 6: Ideal functionality $\mathcal{G}_{\text{att}}^{\text{MAC}}$ with MAC- and signature-based attestation

6.1 Interfaces of $\mathcal{G}_{\text{att}}^{\text{MAC}}$

Initialization. In contrast to $\mathcal{G}_{\text{att}}$, where the adversary provides the signing key to $\mathcal{G}_{\text{att}}$, $\mathcal{G}_{\text{att}}^{\text{MAC}}$ generates the master key pair (vk, sk) for signatures itself during

initialization. The signing key of this key pair is then used to generate attestations and to authenticate keys for designated verifiers. An additional interface, `Getvk`, allows any party to retrieve the public verification key.

Register. `Register` is an additional interface of $\mathcal{G}_{\text{att}}^{\text{MAC}}$ that parties hosting an enclave can call directly. Before an enclave of a party can add attestations for a designated verifier, the verifier must be registered at $\mathcal{G}_{\text{att}}^{\text{MAC}}$. Once registration is complete, both the registered party and the enclave hold a symmetric key resulting from a Diffie-Hellman key exchange. The `Register` interface takes the verifier's share of the key as input and outputs the enclave's share of the key, as well as a signature over both key parts and other parameters. A party hosting an enclave can register any party without an enclave.[5]

Adding an Attestation. Attestation identifiers (AIDs), similar to AIDs in $\mathcal{G}_{\text{att}}$, consist of a uniformly random part and an output from an ITM. Instead of allowing the adversary to choose the attestation algorithms that are used to create aid_2, $\mathcal{G}_{\text{att}}^{\text{MAC}}$ is parameterized with attestation algorithms for both MACs and digital signatures. aid_2 therefore corresponds to the output of the signing algorithm computed within `AddAttestation` or `AddAttestationAll`, respectively. The resulting *aid* is then stored within $\mathcal{G}_{\text{att}}^{\text{MAC}}$ and does not change when further `AddAttestation` requests are made for the same output.

To create attestations for designated verifiers via `AddAttestation`, the verifier has to be registered. This is done via the `Register` interface (see above). After registration, `AddAttestation` can be carried out using the internally stored MAC key. The resume call of $\mathcal{G}_{\text{att}}^{\text{MAC}}$ is identical to $\mathcal{G}_{\text{att}}$.

Verifying an Attestation. In comparison to $\mathcal{G}_{\text{att}}$, `VerifyAttestation` and `VerifyAttestationAll` do not exist in $\mathcal{G}_{\text{att}}^{\text{MAC}}$. Any party that holds a MAC key or the (asymmetric) verification key can verify an attestation. The verification process is therefore performed directly by parties involved, reflecting verification processes in systems with secure hardware.

6.2 Relation to Mechanisms Used in SGX.

The mechanism used for `AddAttestation` in $\mathcal{G}_{\text{att}}^{\text{MAC}}$ is closely related to the mechanisms used by SGX for local attestations, *i.e.* attestations that are transferred between enclaves running on the same system. These use a MAC key to generate attestations that are only useful for a local enclave. To enable remote attestations, a mechanism that is similar to `AddAttestationAll` is used. Using the SGX DCAP SDK, a local enclave is installed that generates a signature key pair

[5] As already discussed in Sect. 5, a corrupted enclave host can impersonate honest parties and register in their name. This is not an issue since it would cause the honest party to be left out of the protocol, not producing any outputs to the environment. Any attestation the impersonated party would obtain this party cannot verify.

and can then be used to turn local attestations into remote ones. However, this system uses a signature key pair per SGX processor. The process to obtain vk on a remote system and to verify it, is more involved for SGX, as it requires communication with an Intel-provided service. While the cryptographic building blocks are similar, we do not model local attestations between enclaves but designated-verifier attestations for parties. However, the interface could be easily extended to this end.

6.3 Relationship between $\mathcal{G}_{\text{att}}$ and $\mathcal{G}_{\text{att}}^{\text{MAC}}$

Wrapper. To show that $\mathcal{G}_{\text{att}}^{\text{MAC}}$ UC-realizes $\mathcal{G}_{\text{att}}$, we need a wrapper protocol which we refer to as $\mathcal{W}$ and present in Fig. 7. From the party's perspective, $\mathcal{W}$ removes interfaces that $\mathcal{G}_{\text{att}}$ has but $\mathcal{G}_{\text{att}}^{\text{MAC}}$ does not and adds interfaces that $\mathcal{G}_{\text{att}}^{\text{MAC}}$ has but $\mathcal{G}_{\text{att}}$ does not. The wrapper of the host party, upon creating the first attestation for a designated verifier, performs a key exchange with the designated verifier's wrapper and the enclave. The verifying party's key is stored inside its wrapper. For subsequent attestation for this verifier, the attestation can be created by $\mathcal{G}_{\text{att}}^{\text{MAC}}$ without key exchange. Since $\mathcal{G}_{\text{att}}^{\text{MAC}}$ lacks interfaces to verify attestations, the wrapper parses these calls by verifying them with the respective verification key, the MAC key in case of designated-verifier attestations or the verification key in case of publicly verifiable attestations. In the case of desginated-verifier attestations, if no MAC key is stored in $\mathcal{W}$, it returns 0.

Shell Program. The shell program (Fig. 8) intercepts the installed program and adds MAC keys at $\mathcal{G}_{\text{att}}^{\text{MAC}}$ via the registration interface. It triggers the registration of a verifier and lets the party afterwards continue the program where it was intercepted before. The shell is agnostic to the installed enclave program.

The role of the wrapper protocol and shell program is illustrated in Fig. 9. While the shell program modifies the interfaces of the enclave program prog, the protocol parties now participate in the wrapper protocol as well.

To show the security of $\mathcal{G}_{\text{att}}^{\text{MAC}}$, we prove that $\mathcal{W}$ UC-realizes $\mathcal{G}_{\text{att}}$.

Theorem 8. *Assuming $\Pi = (\mathsf{KeyGen}, \mathsf{Mac}, \mathsf{Verify})$ is an sEUF-CMA secure MAC scheme and $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is an sEUF-CMA signature scheme, and the DDH assumption holds, $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], p]$ UC-realizes $\mathcal{G}_{\text{att}}$ in the presence of the global functionality $\mathcal{G}_{\text{att}}^{\text{MAC}}$ under static corruption.*

In this proof, we construct a simulator, that, for any enclave program, translates between $\mathcal{G}_{\text{att}}^{\text{MAC}}$ and $\mathcal{G}_{\text{att}}$. In general, the simulator does not let parties install enclaves on $\mathcal{G}_{\text{att}}^{\text{MAC}}$ but instead installs enclaves on $\mathcal{G}_{\text{att}}$. Any operation that cannot be performed by the simulator is forwarded to $\mathcal{G}_{\text{att}}$. The simulator uses the interfaces of $\mathcal{G}_{\text{att}}$ to upload attestation algorithms and a signing key. Note that the simulator must use the algorithms Π and Σ of $\mathcal{G}_{\text{att}}^{\text{MAC}}$ with the modification that, instead of signing prog, they must sign $W[\mathsf{prog}]$ to ensure that corrupt parties can verify attestations in the ideal world while using the respective verification key. For creating designated-verifier attestations for corrupt parties, $\mathcal{G}_{\text{att}}$ requests a MAC key from the simulator, which then performs

Protocol $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P]$

Let $\mathsf{key} = \emptyset$. Let $\mathsf{registered} = \emptyset$.

1. On message initialize from a party P, send initialize to $\mathcal{G}_{\text{att}}^{\text{MAC}}$, call $\mathcal{G}_{\text{att}}^{\text{MAC}}$.`Getvk`(), receive and store vk.
2. On message `Install`(idx, prog) from P, forward call `Install`($idx, W[\mathsf{prog}]$) to $\mathcal{G}_{\text{att}}^{\text{MAC}}$. Return eid to P.
3. On message ("`register`", P_j) from $\mathcal{G}_{\text{att}}^{\text{MAC}}$ for host party $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P]$:
   - If $\mathsf{registered}[eid, P_j] = \bot$
     - Forward ("`register`", eid, P_j) to remote party $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P_j]$.
     - Receive (eid, m) from $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P_j]$.
     - Call $\mathcal{G}_{\text{att}}^{\text{MAC}}$.`Register`$(eid, P_j, m)$, receive (g^b, σ).
     - Send (eid, g^b, σ) to $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P_j]$.
     - Receive ("`registered`", eid) from $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P_j]$.
     - Let $\mathsf{registered}[eid, P_j] = 1$.
   - Call $\mathcal{G}_{\text{att}}^{\text{MAC}}$.`Resume`$(eid,$ ("`continueAttest`", P_j)).
4. On message ("`register`", eid, P) from a host party $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P_i]$ for remote party $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P]$:
   - If $\mathsf{key}[eid] = \bot$
     - Draw and store $a \xleftarrow{\$} \mathbb{Z}_p$.
     - Send (eid, g^a) to $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P_i]$.
     - Receive (eid, g^b, σ).
     - If $\Sigma.\mathsf{Verify}(vk,$ ("`register`", $p, eid, (g^a, g^b)), \sigma) = 1$, let $\mathsf{key}[eid] := (g^b)^a$.
   - Send ("`registered`", eid) to $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], P_i]$. Return out to P.
5. On message `Resume`(eid, inp) from P, forward call to $\mathcal{G}_{\text{att}}^{\text{MAC}}$. Return out to P.
6. On message `AddAttestationAll`$(eid, \mathsf{out}, \mathsf{prog})$ from an ideal functionality, forward call `AddAttestationAll`$(eid, \mathsf{out}, W[\mathsf{prog}])$ to $\mathcal{G}_{\text{att}}^{\text{MAC}}$. Return aid to ideal functionality.
7. On message `VerifyAttestation`$(idx, \mathsf{prog}, eid, aid_1 || aid_2, \mathsf{out})$ from P, if $\mathsf{key}[eid] = \bot$, return 0, else return $aid_2 \stackrel{?}{=} \Pi.\mathsf{Mac}(\mathsf{key}[eid],$ ("`attest`", idx, $W[\mathsf{prog}], eid, aid_1, \mathsf{out}$)).
8. On message `VerifyAttestationAll`$(idx, \mathsf{prog}, eid, aid_1 || aid_2, \mathsf{out})$ from P, return $\Sigma.\mathsf{Verify}(vk,$ ("`attest`", $idx, W[\mathsf{prog}], eid, aid_1, \mathsf{out}), aid_2$).

Fig. 7: Wrapper protocol using $\mathcal{G}_{\text{att}}^{\text{MAC}}$ as a subroutine and UC-realizing $\mathcal{G}_{\text{att}}$

the registration in the real world. This mechanism enforces that $\mathcal{G}_{\text{att}}$ creates attestations on values unknown to the simulator while ensuring that corrupted parties hold the respective verification keys. The signing key to create publicly verifiable attestations is chosen by the adversary to perform a plausible key exchange with corrupted parties in the name of $\mathcal{G}_{\text{att}}^{\text{MAC}}$. Honest parties will always use the interface of $\mathcal{G}_{\text{att}}$ to verify attestations. Corrupted parties, having received a MAC key and/or a verification key from the simulator, can verify attestations themselves successfully. We defer the proof of Thm. 8 to App. I.

Shell $W[\mathsf{prog}]$

On input (inp):
- Step through execution of prog on inp
  - For instruction i, if $i = \texttt{AddAttestation}(P_j, \text{out})$, return $(\text{“}\texttt{register}\text{”}, P_j)$.
  - Else, allow prog to execute i.

On input $(\text{“}\texttt{continueAttest}\text{”}, P_j)$:
- Call $\mathcal{G}_{\text{att}}^{\text{MAC}}.\texttt{AddAttestation}(P_j, \text{out})$ for previously stored values.
- Resume stepping through execution of prog at instruction $i + 1$.

Fig. 8: Shell for enclave program prog

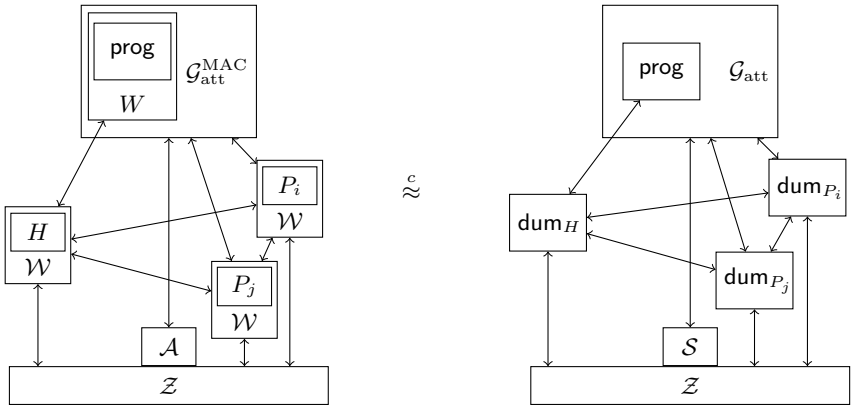


Fig. 9: Illustration of Thm. 8. Dummy parties are denoted with dum_P.

7 Conclusion

We defined an abstraction for attested execution with global public verifiability in the UCGS framework. Our model circumvents the impossibility results from previous formal models and achieves composable general MPC with only one enclave. Our abstraction offers global public verifiability as well as designated-verifier attestations. To comply with existing abstractions, we also gave an alternative definition using MACs and digital signatures and showed how $\mathcal{G}_{\text{att}}^{\text{MAC}}$ relates to $\mathcal{G}_{\text{att}}$.

Acknowledgments. This work was funded by the Topic Engineering Secure Systems of the Helmholtz Association (HGF) and supported by KASTEL Security Research Labs, Karlsruhe.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

[Bad+20] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. "Universal Composition with Global Subroutines: Capturing Global Setup Within Plain UC". In: *TCC 2020: 18th Theory of Cryptography Conference, Part III*. Ed. by Rafael Pass and Krzysztof Pietrzak. Vol. 12552. Lecture Notes in Computer Science. Durham, NC, USA: Springer, Cham, Switzerland, 2020, pp. 1–30. DOI: 10.1007/978-3-030-64381-2_1.

[Bal+20] Foteini Baldimtsi, Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. "Crowd Verifiable Zero-Knowledge and End-to-End Verifiable Multiparty Computation". In: *Advances in Cryptology – ASIACRYPT 2020, Part III*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12493. Lecture Notes in Computer Science. Daejeon, South Korea: Springer, Cham, Switzerland, 2020, pp. 717–748. DOI: 10.1007/978-3-030-64840-4_24.

[Bay+27] Saskia Bayreuther, Robin Berger, Felix Dörre, Eva Hetzel, Yufan Jiang, Christian Martin, Jeremias Mechler, and Jörn Müller-Quade. "More from Less: Composable General Multi-party Computation with Global Public Verifiability from a Single Enclave Only". In: *TCC 2026: 24th Theory of Cryptography Conference, Part IV*. Ed. by Dana Dachman-Soled and Daniel Wichs. Vol. 17194. Lecture Notes in Computer Science. New York City, NY, USA: Springer, Cham, Switzerland, 2027, pp. 328–361. DOI: 10.1007/978-3-032-43630-6_11.

[BDD22] Carsten Baum, Bernardo David, and Rafael Dowsley. "(Public) Verifiability for Composable Protocols Without Adaptivity or Zero-Knowledge". In: *ProvSec 2022: 16th International Conference on Provable Security*. Ed. by Chunpeng Ge and Fuchun Guo. Vol. 13600. Lecture Notes in Computer Science. Nanjing, China: Springer, Cham, Switzerland, 2022, pp. 249–272. DOI: 10.1007/978-3-031-20917-8_17.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. "Publicly Auditable Secure Multi-Party Computation". In: *SCN 14: 9th International Conference on Security in Communication Networks*. Ed. by Michel Abdalla and Roberto De Prisco. Vol. 8642. Lecture Notes in Computer Science. Amalfi, Italy: Springer, Cham, Switzerland, 2014, pp. 175–196. DOI: 10.1007/978-3-319-10879-7_11.

[Bha+21] Pramod Bhatotia, Markulf Kohlweiss, Lorenzo Martinico, and Yiannis Tselekounis. "Steel: Composable Hardware-Based Stateful and Randomised Functional Encryption". In: *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Juan Garay. Vol. 12711. Lecture Notes in Computer Science. Virtual Event: Springer, Cham, Switzerland, 2021, pp. 709–736. DOI: 10.1007/978-3-030-75248-4_25.

[BZ26] Carsten Baum and Chiara-Marie Zok. *Efficient, UC-secure and Publicly Auditable MPC from OLE & VOLE-in-the-head*. Cryptology

ePrint Archive, Report 2026/337. 2026. URL: https://eprint.iacr.org/2026/337.

[Can01] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd Annual Symposium on Foundations of Computer Science*. Las Vegas, NV, USA: IEEE Computer Society Press, 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[Can+02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. "Universally composable two-party and multi-party secure computation". In: *34th Annual ACM Symposium on Theory of Computing*. Montréal, Québec, Canada: ACM Press, 2002, pp. 494–503. DOI: 10.1145/509907.509980.

[Can+07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. "Universally Composable Security with Global Setup". In: *TCC 2007: 4th Theory of Cryptography Conference*. Ed. by Salil P. Vadhan. Vol. 4392. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer Berlin Heidelberg, Germany, 2007, pp. 61–85. DOI: 10.1007/978-3-540-70936-7_4.

[Can20] Ran Canetti. "Universally Composable Security". In: *J. ACM* 67.5 (2020), pp. 1–94. DOI: 10.1145/3402457.

[CC25] Ran Canetti and Megan Chen. "Universally Composable Succinct Vector Commitments and Applications". In: *TCC 2025: 23rd Theory of Cryptography Conference, Part IV*. Ed. by Benny Applebaum and Huijia (Rachel) Lin. Vol. 16271. Lecture Notes in Computer Science. Aarhus, Denmark: Springer, Cham, Switzerland, 2025, pp. 642–674. DOI: 10.1007/978-3-032-12290-2_21.

[CD16] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. 2016. URL: https://eprint.iacr.org/2016/086.

[CDL16] Jan Camenisch, Manu Drijvers, and Anja Lehmann. "Universally Composable Direct Anonymous Attestation". In: *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Vol. 9615. Lecture Notes in Computer Science. Taipei, Taiwan: Springer Berlin Heidelberg, Germany, 2016, pp. 234–264. DOI: 10.1007/978-3-662-49387-8_10.

[CF01] Ran Canetti and Marc Fischlin. "Universally Composable Commitments". In: *Advances in Cryptology – CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, 2001, pp. 19–40. DOI: 10.1007/3-540-44647-8_2.

[Che+24] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. "Intel TDX Demystified: A Top-Down Approach". In: *ACM Comput. Surv.* 56.9 (2024), 238:1–238:33. DOI: 10.1145/3652597. URL: https://doi.org/10.1145/3652597.

[CSV16] Ran Canetti, Daniel Shahaf, and Margarita Vald. "Universally Composable Authentication and Key-Exchange with Global PKI". In: *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Vol. 9615. Lecture Notes in Computer Science. Taipei, Taiwan: Springer Berlin Heidelberg, Germany, 2016, pp. 265–296. DOI: 10.1007/978-3-662-49387-8_11.

[DMM23] Felix Dörre, Jeremias Mechler, and Jörn Müller-Quade. "Practically Efficient Private Set Intersection from Trusted Hardware with Side-Channels". In: *Advances in Cryptology – ASIACRYPT 2023, Part IV*. Ed. by Jian Guo and Ron Steinfeld. Vol. 14441. Lecture Notes in Computer Science. Guangzhou, China: Springer, Singapore, Singapore, 2023, pp. 268–301. DOI: 10.1007/978-981-99-8730-6_9.

[Fis+17] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. "IRON: Functional Encryption using Intel SGX". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, 2017, pp. 765–782. DOI: 10.1145/3133956.3134106.

[GKR23] Mike Graf, Ralf Küsters, and Daniel Rausch. "AUC: Accountable Universal Composability". In: *2023 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, 2023, pp. 1148–1167. DOI: 10.1109/SP46215.2023.10179384.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *19th Annual ACM Symposium on Theory of Computing*. Ed. by Alfred Aho. New York City, NY, USA: ACM Press, 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[Mis+25] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. "Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX". In: *Abstracts of the 2025 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2025, Stony Brook, NY, USA, June 9-13, 2025*. Ed. by Anshul Gandhi, Zhenhua Liu, Sewoong Oh, Ramesh K. Sitaraman, and Benny Van Houdt. ACM, 2025, pp. 34–36. DOI: 10.1145/3726854.3727280. URL: https://doi.org/10.1145/3726854.3727280.

[MK25] Lorenzo Martinico and Markulf Kohlweiss. "AGATE: Augmented Global Attested Trusted Execution in the Universal Composability Framework". In: *CSF 2025: IEEE 38th Computer Security Foundations Symposium*. Santa Cruz, CA, USA: IEEE Computer Society Press, 2025, pp. 49–64. DOI: 10.1109/CSF64896.2025.00008.

[PST17] Rafael Pass, Elaine Shi, and Florian Tramèr. "Formal Abstractions for Attested Execution Secure Processors". In: *Advances in Cryptology*

– *EUROCRYPT 2017, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. Paris, France: Springer, Cham, Switzerland, 2017, pp. 260–289. DOI: 10.1007/978-3-319-56620-7_10.

[RHK25] Daniel Rausch, Nicolas Huber, and Ralf Kuesters. *Verifiable E-Voting with a Trustless Bulletin Board*. Cryptology ePrint Archive, Report 2025/841. 2025. URL: https://eprint.iacr.org/2025/841.

[SFF20] Muhammad Usama Sardar, Rasha Faqeh, and Christof Fetzer. "Formal Foundations for Intel SGX Data Center Attestation Primitives". In: *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*. Ed. by Shang-Wei Lin, Zhe Hou, and Brendan P. Mahony. Lecture Notes in Computer Science. Springer, 2020, pp. 268–283. DOI: 10.1007/978-3-030-63406-3_16. URL: https://doi.org/10.1007/978-3-030-63406-3_16.

[Sub+17] Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. "A Formal Foundation for Secure Remote Execution of Enclaves". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, 2017, pp. 2435–2450. DOI: 10.1145/3133956.3134098.

[Tra+17] Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. "Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge". In: *2017 IEEE European Symposium on Security and Privacy*. Paris, France: IEEE Computer Society Press, 2017, pp. 19–34. DOI: 10.1109/EuroSP.2017.28.

[Yao86] Andrew Chi-Chih Yao. "How to Generate and Exchange Secrets (Extended Abstract)". In: *27th Annual Symposium on Foundations of Computer Science*. Toronto, Ontario, Canada: IEEE Computer Society Press, 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

A Introduction to UC

In the following, we continue the overview of UC from Sect. 2.2, which is adapted from [DMM23].

UC Framework Conventions. In the UC framework, each party is identified by its party identifier (PID) *pid* which is unique to the party and is the UC equivalent of the physical identity of this party. A party running a protocol π is called the *main party* of this instance of π. Let μ denote the code of parties of π. A subsidiary and its parent use their *input/subroutine output* tape to communicate with each other. The set of machines taking part in the same protocol but for different parties communicate through their *backdoor* tapes via the adversary. An instance of a protocol is identified by its session identifier (SID) *sid*. All machines taking part in the same protocol instance share the same SID. A specific machine is identified uniquely by its extended identity (sid, μ, pid).

The (Dummy) Adversary. The adversary $\mathcal{A}$ is instructed by the environment and represents the environment's interface to the execution. To this end, all messages from any party to a party that has a different main party and that are intended to be written to an *incoming message* tape are copied to the adversary. The adversary can process the message arbitrarily. The adversary may decide to deliver the message (by writing the message on its own outgoing message tape), postpone or completely suppress the message, inject new messages or alter messages in any way including the recipient and/or alleged sender.

The environment may also instruct the adversary to corrupt a party. In this case, the adversary takes over the position of the corrupted party, reports its internal state to the environment and from then on may arbitrarily deviate from the protocol in the name of the corrupted party (as requested by the environment). This means whenever the corrupted machine would have been activated (even due to subroutine output), the adversary gets activated with the same input.

A special case for the adversary is the so-called *dummy adversary* that reports all received messages to the environment and delivers all messages coming from the environment. It can be shown that the dummy adversary is *complete*, *i.e.* that if a simulator for the dummy adversary exists, then there also exists a simulator for any other adversary.

Ideal Functionalities and the Ideal Protocol. An ideal functionality $\mathcal{F}$ is a special type of entity whose instances bear an SID but no PID. Hence, it is an exception to the aforementioned identification scheme. Input to and subroutine output from $\mathcal{F}$ are performed through dummy parties. Dummy parties merely forward their input to the input tape of $\mathcal{F}$ and subroutine output from $\mathcal{F}$ to their own outgoing message tape. They share the same SID as $\mathcal{F}$ but additionally have individual party identifiers (PIDs) as if they were the actual main parties of a (real) protocol. The ideal functionality $\mathcal{F}$ is simultaneously a subroutine for each

dummy party and conducts the prescribed task. IDEAL($\mathcal{F}$) is called the *(ideal) protocol* for $\mathcal{F}$ and denotes the set of $\mathcal{F}$ together with its dummy parties.

The UC Experiment. Let π be a protocol, $\mathcal{Z}$ an environment and $\mathcal{A}$ an adversary. Let ξ be an (efficiently computable) identity bound. The UC experiment, denoted by $\mathsf{Exec}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, a)$, initially activates the environment $\mathcal{Z}$ with security parameter κ and input $a \in \{0,1\}^*$. The first machine that is invoked by $\mathcal{Z}$ is the adversary $\mathcal{A}$. All other parties invoked by $\mathcal{Z}$ are set to be main parties of the challenge protocol π. $\mathcal{Z}$ freely chooses their input, their PIDs and the SID of the challenge protocol. When giving input, the environment is restricted to using extended identities such that ξ is satisfied at every point of the execution. The experiment is executed as outlined above.

Definition of Security. Let π, ϕ be protocols. *π emulates ϕ in the UC framework*, denoted by $\pi \geq \phi$ with respect to ξ-bounded environments, if for every PPT adversary $\mathcal{A}$ there is a PPT adversary $\mathcal{S}$ such that for every ξ-identity-bounded PPT environment $\mathcal{Z}$, there is a negligible function negl such that for all $\kappa \in \mathbb{N}, a \in \{0,1\}^*$ it holds that

$$|\Pr[\mathsf{Exec}\big(\pi, \mathcal{A}, \mathcal{Z}\big)(\kappa, a) = 1] - \Pr[\mathsf{Exec}\big(\phi, \mathcal{S}, \mathcal{Z}\big)(\kappa, a) = 1]| \leq \mathsf{negl}(\kappa)$$

where $\mathsf{Exec}\big(\pi, \mathcal{A}, \mathcal{Z}\big)(\kappa, a)$ denotes the random variable for the environment $\mathcal{Z}$'s output in the UC execution experiment with protocol π and adversary $\mathcal{A}$ on input a and security parameter κ.

The simulator $\mathcal{S}$ mimics the adversarial behavior to the environment as if this were the real experiment with real parties carrying out the real protocol with real π-messages. Moreover, $\mathcal{S}$ must come up with a convincing internal state upon corrupted parties, consistent with the simulated protocol execution up to this point (dummy parties do not have an internal state).

Protocol Composition. UC security is closed under protocol composition: Let π, ϕ, be subroutine-respecting protocols[6] and let ρ be an arbitrary protocol.

$$\pi \geq \phi \implies \rho^{\pi} \geq \rho^{\phi}$$

"Await" in Protocols. When specifying our protocols, we use the keyword "await" to indicate that a party waits for a message from another party before continuing with the protocol. Due to the nature of activation in UC, care has to be taken when Turing Machines await a message. When a party performs a computation and then waits for a message from another party, it must give up activation, so the other party can get scheduled and then be able to send a message. If a party gives up activation without sending a message to a different

[6] Very informally, a protocol π is subroutine-respecting if a machine μ of π does not communicate with a machine μ' that is not part of π or of a sub-protocol of π.

party, the environment gets scheduled, which can then decide to schedule the party we await a message from, but it can also decide not to do that. This might have implications on the indistinguishability of the real and ideal execution.

As we consider this to mainly be an artifact of the asynchronous nature of UC, we will not go into detail on this in our main paper. However, we want to emphasize, that this issue can be generically solved, by using a subprotocol, which determines the next party, which needs to be activated for the main protocol to progress, and then sends it a message, such that this party gets activated. Instead of awaiting a specific input, a party would then execute this subprotocol. If its input does not match the one it is awaiting, it would also execute this subprotocol.

One way to do this is for a party to check whether it is the one which needs to perform the next step in the protocol. If it is, it will do so. Otherwise, it will send a subprotocol-message to the lexicographically next party, which will then either continue execution, or it will itself send a message to the lexicographically next party.

B Ciphertext Integrity (INT-CTXT)

Informally, ciphertext integrity captures the property that an adversary, given an encryption oracle and a verification oracle, cannot create a new ciphertext that can be decrypted successfully. A new ciphertext denotes a ciphertext that the adversary did not obtain from the encryption oracle. We consider the following INT-CTXT experiment for an adversary $\mathcal{A}$ and a security parameter κ.

$\mathsf{Exp}^{\mathsf{INT\text{-}CTXT}}_{\mathsf{AE},\mathcal{A}}(\kappa, z)$	$\mathsf{Enc}(k, m)$	$\mathsf{Verify}(k, c)$
$\mathsf{L} := \emptyset$	$c \leftarrow \mathsf{AE.Enc}(k, m)$	**return** $\mathsf{AE.Dec}(k, c) \neq \bot$
$k \xleftarrow{\$} \mathcal{K}$	$\mathsf{L} := \mathsf{L} \cup \{c\}$	
$c^* \leftarrow \mathcal{A}^{\mathsf{Enc}(k,\cdot),\mathsf{Verify}(k,\cdot)}(1^\kappa, z)$	**return** c	
$m^* := \mathsf{Dec}(k, c^*)$		
return $m^* \neq \bot \wedge c^* \notin \mathsf{L}$		

Definition 7. *INT-CTXT Security for Symmetric Encryption* We say that a symmetric encryption scheme $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ satisfies *ciphertext integrity* or is *INT-CTXT secure* if for every PPT adversary $\mathcal{A}$, there exists a negligible function negl such that for every $\kappa \in \mathbb{N}$ and every $z \in \{0,1\}^*$:

$$\Pr[\mathrm{Exp}^{\mathsf{INT\text{-}CTXT}}_{\mathsf{AE},\mathcal{A}}(\kappa, z) = 1] \leq \mathsf{negl}(\kappa)$$

where the probability is over the random coins of $\mathcal{A}$ and $\mathrm{Exp}^{\mathsf{INT\text{-}CTXT}}_{\mathsf{AE},\mathcal{A}}$.

C The Role of *idx* in $\mathcal{G}_{\text{att}}$

In our ideal functionality $\mathcal{G}_{\text{att}}$, when installing an enclave, the enclave host has to input an index idx in addition to the enclave program. For honest parties, $\mathcal{G}_{\text{att}}$ verifies that they input their session ID as idx, otherwise $\mathcal{G}_{\text{att}}$ aborts the installation process. Without this check, a protocol could be designed in such a way that an honest party would install an enclave in a session that differs from its own. Without any check to tie the enclave ID (eid) to the host's session, honest parties from different sessions would now interact with the same enclave. Since all parties are honest, this could lead to extractability issues during the simulation. In case of corruptions, extraction is easy and therefore the check is only necessary for honest parties.

D General MPC Continued

As described in Sect. 4, the ideal functionality presented in Def. 2 can implement a variety of local ideal functionalities. To give two examples of how f can be specified, Example 1 shows what a commitment functionality can look like in this notation, while Example 2 shows a coin toss.

The state is always set to ε at the first activation of $\mathcal{F}_f$. When receiving an input inp, *e.g.*, $(\texttt{commit}, m)$, the state is updated depending on the current state, the randomness r and the party that gave the input (*e.g.*, the committer C). In addition to the state, the party receiving the output (*e.g.*, R) and the output value out itself (*e.g.*, $\texttt{committed}$) are computed by $\mathcal{F}_f$ as well. In Example 1, the state after the commit phase is $(\text{“}\texttt{committed}\text{”}, m)$ and the committer can now decide to unveil the commitment with another input $\texttt{unveil}$.

Example 1. For a commitment scheme with committer C and receiver R, f could look as follows:

$$f(\text{state}, r, p, \mathsf{inp}) = \begin{cases} ((\text{“}\texttt{committed}\text{”}, m), R, \texttt{committed}) & (\varepsilon, \cdot, C, (\texttt{commit}, m)) \\ (\text{“}\texttt{unveiled}\text{”}, R, (\texttt{unveiled}, m)) & ((\text{“}\texttt{committed}\text{”}, m), \cdot, C, \texttt{unveil}) \\ (\text{state}, \bot, \varepsilon) & \text{otherwise} \end{cases}$$

Using this f, the ideal functionality $\mathcal{F}_f$ behaves exactly as $\mathcal{F}_{\text{COM}}$.

Example 2. For a coin toss between two parties $\mathcal{P} = \{P_1, P_2\}$, f could look as follows:

$$f(\text{state}, r, p, x) = \begin{cases} (r_0, P_i, r_0) & (\varepsilon, \cdot, P_i, \texttt{coinToss}) \\ (\text{state}, P_i, \text{state}) & (\text{state}, \cdot, P_i, \texttt{coinToss}) \\ (\text{state}, \bot, \varepsilon) & \text{otherwise} \end{cases}$$

Using this f, the ideal functionality $\mathcal{F}_f$ behaves exactly as $\mathcal{F}_{\text{CT}}$.

E Proof of Theorem 5 (General MPC)

In the following, we prove that with Constr. 1, we can realize the ideal functionality $\mathcal{F}_f$, therefore being able to construct general MPC using only a single enclave.

Theorem 5 *Assuming* AE *is an INT-CTXT, IND-CPA secure, perfectly correct encryption scheme and the DDH assumption holds,* π_f *UC-realizes* $\mathcal{F}_f$ *in the presence of* $\mathcal{G}_{\text{att}}$ *under static corruption.*

Proof. **Definition 8.** *Simulator* $\mathcal{S}$ If the host is honest, process all following input messages sent to the host according to the protocol while still intercepting interactions of the host with $\mathcal{G}_{\text{att}}$, as if it was corrupted.

In the following description of the simulator, items are numbered, so they can be referenced in the proof. They are, however, not limited to be called in this order.

1. On first activation:
   - Draw $eid \xleftarrow{\$} \{0,1\}^\kappa$.
   - Initialize $\mathsf{L} = \emptyset$ and $\mathsf{key}_{\mathsf{aid}} \coloneqq \emptyset$.
   - Initialize $\mathsf{oids}_P = \emptyset$ for all $P \in \mathcal{P}$.
2. On any backdoor message `Initialize` from $\mathcal{G}_{\text{att}}$, or message for enclaves not within this session, forward the message to the dummy adversary and forward the response back to $\mathcal{G}_{\text{att}}$.
3. On call $\mathcal{G}_{\text{att}}.\texttt{Install}(idx, \mathsf{prog}')$ from the corrupted host H:
   - If $\mathsf{prog}' = \mathsf{prog}_f$:
     - Do not forward the call.
     - Send eid to H in the name of $\mathcal{G}_{\text{att}}$.

     Else:
     - Forward the `Install` call to $\mathcal{G}_{\text{att}}$ and await eid'.
     - Forward all subsequent calls from corrupted parties to $\mathcal{G}_{\text{att}}$ regarding this eid'.
     - Send eid' to H in the name of $\mathcal{G}_{\text{att}}$.
4. For each honest party P in $\mathcal{P}$:
   - Draw $x_P \xleftarrow{\$} \mathbb{Z}_p$.
   - Set $X_P \coloneqq g^{x_P}$.
   - Send ("`keyExchange`", X_P) from P to H and await ("`keyExchange`", eid', aid', Y'_P) from H, let P abort, if (X_P, Y'_P) has not been the output of $\mathcal{S}$ (in Item 5) for P or if (aid') has not been the corresponding attestation generated by $\mathcal{S}$ (in Item 6).
   - Send message ("`eid`", eid') from P to every other party.
   - Await message ("`eid`", $eid''_{P'}$) from every other party P' to P: If $eid''_{P'} \neq eid'$, for any P', let P abort.

// Consider the following cases only when eid is the id of an enclave with the correct program prog_f

5. On call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{"}\texttt{keyExchange}\text{"}, P, X'_P))$ from host H:
   - Do not forward the call.
   - Draw $y_P \xleftarrow{\$} \mathbb{Z}_p$ and set $Y_P \coloneqq g^{y_P}$.

- If P is honest and $X'_P \stackrel{?}{=} X_P$, set $k_P \xleftarrow{\$} \mathcal{K}$. Otherwise, set $k_P := (X'_P)^{y_P}$.
- Send (X'_P, Y_P) to H in the name of $\mathcal{G}_{\text{att}}$.

6. On call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{“}\texttt{attest}\text{”}, P))$ from corrupted host H:
   - Do not forward the call.
   - Draw $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$. If $\mathsf{key}_{\mathsf{aid}}[P, eid] = \bot$, ask the adversary for a key and set $\mathsf{key}_{\mathsf{aid}}[P, eid]$ to this key if P is corrupted. If P is honest, set $\mathsf{key}_{\mathsf{aid}}[P, eid] \leftarrow \Pi.\mathsf{KeyGen}(1^\kappa)$ or, if the environment returned $\bot$ to the request for attestation algorithms, set $\mathsf{key}_{\mathsf{aid}}[P, eid] = 0^\kappa$.
   - Compute $aid_2 = \Pi.\mathsf{Mac}(\mathsf{key}_{\mathsf{aid}}[P, eid], (\text{“}\texttt{attest}\text{”}, sid, \mathsf{prog}_f, eid, aid_1, (X'_P, Y_P)))$.
   - Send $aid_P := aid_1 || aid_2$ to H in the name of $\mathcal{G}_{\text{att}}$.
   - For all subsequent calls $\mathcal{G}_{\text{att}}.\texttt{VerifyAttestation}(idx, \mathsf{prog}_f, eid, aid_P, (X_P, Y_P))$ from P, return 1.
7. On backdoor message (eid, H) and request for key k' for P from $\mathcal{G}_{\text{att}}$, send $k' := \mathsf{key}_{\mathsf{aid}}[P, eid]$ back to $\mathcal{G}_{\text{att}}$.
8. On notification of input with length ℓ from honest party P to $\mathcal{F}_f$, generate $c_P^{\text{inp}} \leftarrow \mathsf{AE.Enc}(k_P, 0^\ell)$ and send $(\text{“}\texttt{input}\text{”}, c_P^{\text{inp}})$ from P to H.
9. On call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{“}\texttt{input}\text{”}, P, c'_{\text{inp}}))$ from H to $\mathcal{G}_{\text{att}}$:
   - Do not forward the call.
   - If $c'_{\text{inp}} \in \mathsf{L}$, ignore this call and all future $\mathcal{G}_{\text{att}}.\texttt{Resume}$ calls regarding this eid.
   - Set $\mathsf{L} := \mathsf{L} \cup \{c'_{\text{inp}}\}$.
   - If P is honest and c'_{inp} has not been the output of $\mathcal{S}$ (in Item 8), abort. Else if P is corrupted:
     - Set $m := \mathsf{AE.Dec}(k_P, c'_{\text{inp}})$.
     - If $m \neq \bot$, input m to $\mathcal{F}_f$ in the name of P and wait for a notification from $\mathcal{F}_f$ about the input.
   - Acknowledge the notification from $\mathcal{F}_f$ about the input and wait for a public delayed output out from $\mathcal{F}_f$ to a party P' or for a backdoor message (“$\texttt{output}$”) from $\mathcal{F}_f$ to $\mathcal{S}$.
   - Sample $\nu \xleftarrow{\$} \{0,1\}^\kappa$.
   - In case of a public delayed output, output (ν, P', out) to H in the name of $\mathcal{G}_{\text{att}}$. Otherwise, output $(\nu, \bot, \varepsilon)$ to H in the name of $\mathcal{G}_{\text{att}}$.
10. On call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, \text{“}\texttt{attestOutput}\text{”})$ from corrupted host H to $\mathcal{G}_{\text{att}}$:
    - Do not forward the call.
    - If the last output of $\mathcal{S}$ in Item 9 was of the form $(\nu, P', \mathsf{out}) \stackrel{?}{=} (\cdot, \bot, \varepsilon)$, ignore the call.
    - Draw $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$.
    - Generate $aid_2 = \Pi.\mathsf{Mac}(\mathsf{key}_{\mathsf{aid}}[P', eid], (\text{“}\texttt{attest}\text{”}, sid, \mathsf{prog}_f, eid, aid_1, (\nu, P', \mathsf{out})))$.
    - Send $aid := aid_1 || aid_2$ to H in the name of $\mathcal{G}_{\text{att}}$.
    - For all subsequent calls $\mathcal{G}_{\text{att}}.\texttt{VerifyAttestation}(idx, \mathsf{prog}_f, eid, aid, (\nu, P', \mathsf{out}))$ from P', return 1.
11. On message $(\text{“}\texttt{output}\text{”}, aid', \nu', \mathsf{out}')$ from host H to an honest party P:

- If (ν', P, out') was not output by $\mathcal{S}$ in Item 11 or aid' was not output by $\mathcal{S}$ in Item 10, let P abort.
- Set $\mathsf{oids}_P \coloneqq \mathsf{oids}_P \cup \{\nu'\}$.
- Allow the delayed output from $\mathcal{F}_f$.

We show the indistinguishability of the following hybrids. Let out_i denote the output of the environment in hybrid G_i.

G_0: Execution with ideal functionality $\mathcal{F}_0 = \mathcal{F}_f$ and the simulator $\mathcal{S}_0 = \mathcal{S}$.

G_1: Execution with an ideal functionality $\mathcal{F}_1$ that behaves the same as $\mathcal{F}_0$, except that it tells the inputs of honest parties to the simulator. The simulator $\mathcal{S}_1$ behaves the same as $\mathcal{S}_0$, except that in Item 8, it also learns the input m from $\mathcal{F}_1$. It therefore no longer encrypts 0^ℓ for honest parties, but uses their inputs m instead.

G_2: Execution with ideal functionality $\mathcal{F}_2$, that no longer generates outputs based on f and previous inputs but lets the adversary determine the output for all honest parties. The simulator now makes outputs to honest parties based on the internal state of the simulated enclave (evaluating f).

G_3: Execution with an ideal functionality $\mathcal{F}_3 = \mathcal{F}_2$. $\mathcal{S}_3$ no longer aborts in Item 9 if it receives a call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, (\text{“}\texttt{input}\text{”}, P, c))$ from H for an honest party P and and a ciphertext c, which has has not been previously output by $\mathcal{S}_3$.

G_4: Execution with an ideal functionality $\mathcal{F}_4 = \mathcal{F}_3$. In Item 5 the simulator $\mathcal{S}_4$ now also uses the key $k_P \coloneqq (X'_P)^{y_P}$ from the DH key exchange for honest parties P instead of using a random key k_P.

G_5: Execution with an ideal functionality $\mathcal{F}_5 = \mathcal{F}_4$. Instead of intercepting calls to $\mathcal{G}_{\text{att}}$ and simulating the enclave program itself, $\mathcal{S}_5$ now passes through these calls. It no longer performs the computation itself but instead makes or passes through calls to $\mathcal{G}_{\text{att}}$.
This effectively removes Items 3, 5, 6 and 10 and the remaining of Item 9. What remains is a simulator that asks the adversary if it should deliver inputs/outputs, which it then makes through the ideal functionality and executes the protocol on behalf of the honest parties.

G_6: Real execution with π_f, $\mathcal{G}_{\text{att}}$ and the dummy adversary.

Claim 1. out_0 and out_1 are distributed identically.

Proof. Straightforward by reduction to the IND-CPA property of AE using a hybrid argument for all (keys of) honest parties.

Claim 2. out_1 and out_2 are indistinguishable.

Proof. G_1 and G_2 are identical, except that now the evaluation of f determining the output of parties is no longer done by $\mathcal{F}_1$, but instead by the simulated enclave program on the same inputs. Since the same inputs (including randomness) are used, the output of f and therefore the output to honest parties and the output to corrupted parties, that is then further used during simulation is identical to the output in $\mathcal{F}_1$. Therefore, the two games are distributed identically.

Claim 3. Assuming the AE is an INT-CTXT encryption scheme, out_2 and out_3 are computationally indistinguishable.

Proof. Straightforward by reduction to the INT-CTXT property of AE using a hybrid argument for all (keys of) honest parties.

Claim 4. Assuming the DDH problem is hard, out_3 and out_4 are computationally indistinguishable.

Proof. Straightforward by reduction to the DDH assumption using a hybrid argument for all changed keys.

Claim 5. out_4 and out_5 are distributed identically.

Proof. As $\mathcal{G}_{\text{att}}$ does not hold any shared state between enclaves with different *eid*, the simulator only has to simulate $\mathcal{G}_{\text{att}}$'s response for the current session. And as the internal state of $\mathcal{G}_{\text{att}}$ in G_5 can only be observed by parties within the current session, the environment does not see any state update directly through honest parties, but only through the dummy adversary through corrupted parties within this session. As these calls are intercepted by the simulator in G_4 and answered accordingly, out_4 and out_5 are distributed identically.

Claim 6. out_5 and out_6 are distributed identically.

Proof. The difference between these games is only syntactical. Therefore, this claim follows directly from the definition of the hybrid games.

Using Claim 1 to 6, we know that $\mathsf{G}_0 \overset{c}{\approx} \mathsf{G}_6$. The theorem follows.

F Proof of Theorem 6 (General MPC with Publicly Verifiable Output)

In the following, we prove that we can use $\mathcal{G}_{\text{att}}$ to realize $\mathcal{F}_{\mathsf{m}f}$ in a publicly verifiable way. Major differences to the proof of Thm. 5 are highlighted.

Theorem 6 *Assuming* AE *is an INT-CTXT, IND-CPA secure, perfectly correct encryption scheme and the DDH assumption holds, the protocol* $\pi^{\mathsf{pub}}_{\mathsf{mult},f}$ *UC-realizes* $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ *in the presence of* $\mathcal{G}_{\text{att}}$ *and* $\mathcal{G}_{\text{reg}}$ *under static corruption.*

To prove Thm. 6, we define one simulator, which covers all combinations of corrupted and honest parties, including the host.

The simulator defined in Def. 9 works in a similar way as in the non-publicly verifiable case of Thm. 5. Designated-verifier attestations are performed entirely in the simulator's head, without an actual enclave necessary. Only the publicly verifiable attestation has to be valid when using the verification interface at $\mathcal{G}_{\text{att}}$, since the simulator cannot intercept this kind of request. The registration of honest parties at $\mathcal{G}_{\text{reg}}$ and the overwriting of the output value via $\mathcal{G}_{\text{att}}.\texttt{AddAttestationAll}$ are all handled by $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$.

For the case of corrupted enclave hosts, all calls from the host to $\mathcal{G}_{\text{att}}$ have to be intercepted and for honest parties, to verify the host's messages, checks have to be performed locally by the simulator instead of calling $\mathcal{G}_{\text{att}}$.`VerifyAttestation`. Only the $\mathcal{G}_{\text{att}}$.`VerifyAttestationAll` calls for publicly verifiable outputs are let through to $\mathcal{G}_{\text{att}}$.

Proof. We start our proof by defining the simulator.

Definition 9. *Simulator* $\mathcal{S}$

- On first activation, set $\mathsf{X}_{\mathsf{prot}} := \emptyset, \mathsf{X}_{\mathsf{encl}} := \emptyset, \mathsf{Y} := \emptyset$, $\mathsf{s} := \emptyset$, $\mathsf{aid} := \emptyset$, $\mathsf{key} := \emptyset$, $\mathsf{key}_{\mathsf{aid}} := \emptyset$, $\mathsf{c}_{\mathsf{ident}} := \emptyset$, $\mathsf{c}_{\mathsf{inp}} := \emptyset$ and $\mathsf{L} := \emptyset$.
- On backdoor message `Initialize` from $\mathcal{G}_{\text{att}}$, forward to the environment and forward the response from the environment back to $\mathcal{G}_{\text{att}}$.
- Forward backdoor messages ("`registered`", P) from $\mathcal{G}_{\text{reg}}$ to the environment and the acknowledgment back to $\mathcal{G}_{\text{reg}}$.
- On backdoor message ("`setup`", fid, pid_P) from $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$, for each party P' that gets activated, starting at P:
  - Send ("`register`", $fid, pid_{P'}$) to $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$ and wait for a response ("`registered`", $fid, pid_{P'}$)
  - Set $\mathsf{oids}_{P'} := \emptyset$.
  - Draw $x \xleftarrow{\$} \mathbb{Z}_p$ and set $\mathsf{X}_{\mathsf{prot}}[fid, P'] := g^x$.
  - Send ("`keyExchange`", $fid, \mathsf{X}_{\mathsf{prot}}[fid, P']$) from P' to H.
- On message ("`keyExchange`", fid, eid', aid', Y', s') from host H to honest party P:
  - For each party P' with $pid_{P'} \in \mathcal{P} \setminus \{pid_P\}$:
    - Call $\mathcal{G}_{\text{reg}}$.`Retrieve`(P').
    - Await public delayed output ("`retrieve`", $P', vk_{P'}$) from $\mathcal{G}_{\text{reg}}$ that was acknowledged by the environment.
    - Let P abort if $vk_{P'} = \bot$.
  - Let P abort if there is no stored record ("`enclave`", $\cdot, eid', \mathsf{prog}_f[\{(P', vk_{P'})\}_{pid_{P'} \in \mathcal{P}}]$) such that the message from H is ("`keyExchange`", $fid, eid', \mathsf{aid}[P, eid'], \mathsf{Y}[P, eid'], \mathsf{s}[P, eid']$) and such that $\mathsf{X}_{\mathsf{encl}}[P, eid'] = \mathsf{X}_{\mathsf{prot}}[fid, P]$.
  - Store ("`fidEid`", fid, P, eid').
  - Compute $\mathsf{c}_{\mathsf{ident}}[fid, P] \leftarrow \mathsf{AE.Enc}(\mathsf{key}[P, eid'], 0^\ell)$, where ℓ is the length of a signature of a κ-bit string.
  - Send ("`identify`", $fid, \mathsf{c}_{\mathsf{ident}}[fid, P]$) from P to H.
  - Send message ("`eid`", fid, eid') from P to each other party.
  - Await message ("`eid`", fid, eid'') from all other parties to P: If $eid'' \neq eid'$, let P abort.
  - If there was a previous backdoor message ("`setup`", fid, pid_P) from $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$ to $\mathcal{S}$:
    - If $\mathcal{S}$ has not sent ("`chooseEid`", $fid, \cdot$) to $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$ yet, send ("`chooseEid`", fid, eid') to $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$ and await response ("`eid`", fid, eid).
    - Send ("`continue`", fid, pid_P) to $\mathsf{sh}[\mathcal{F}_{\mathsf{mf}}]$.

- On call $\mathcal{G}_{\text{reg}}$.`Register`(vk_P) from corrupted party P, forward the call to $\mathcal{G}_{\text{reg}}$.
- On call $\mathcal{G}_{\text{reg}}$.`Retrieve`(P') from corrupted party P or from corrupted host H, pass through, await public delayed output ("`retrieve`", $P', vk_{P'}$), ask the environment for acknowledgment and forward to P or H, respectively.
- On message ("`keyExchange`", fid, X') from party P to honest host H:
  - Call $\mathcal{G}_{\text{reg}}$.`Retrieve`(P).
  - Await public delayed output ("`retrieve`", P, vk_P) from $\mathcal{G}_{\text{reg}}$ that was acknowledged by the environment. Let H abort if $vk_P = \bot$.
  - Wait until H has received ("`keyExchange`", fid, $\cdot$) from all parties P' with $pid_{P'} \in \mathcal{P}$.
  - If $\mathcal{S}$ has not sent ("`chooseEid`", fid, $\cdot$) to $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ yet:
    - Send ("`candidateEid`") to $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$.
    - Forward delayed output notifications from $\mathcal{G}_{\text{reg}}$ to the environment and the respective responses from the environment back to $\mathcal{G}_{\text{reg}}$.
    - Receive ("`candidateEid`", eid) from $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$.
    - Send ("`chooseEid`", fid, eid) to $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ and await response ("`eid`", fid, eid).
    - Store ("`enclave`", $H, eid, \mathsf{prog}_f[\{(P', vk_{P'})\}_{pid_{P'} \in \mathcal{P}}]$).
  - Draw $y \xleftarrow{\$} \mathbb{Z}_p$ and $\mathsf{s}[P, eid], \mathsf{aid}[P, eid] \xleftarrow{\$} \{0,1\}^\kappa$ and set $\mathsf{X}_{\mathsf{encl}}[P, eid] := X'$ and $\mathsf{Y}[P, eid] := g^y$.
  - If P is honest, set $\mathsf{key}[P, eid] \xleftarrow{\$} \mathcal{K}$. Otherwise, set $\mathsf{key}[P, eid] := (X')^y$.
  - Send ("`keyExchange`", $fid, eid, \mathsf{aid}[P, eid], \mathsf{Y}[P, eid], \mathsf{s}[P, eid]$) from H to P.
  - If P is corrupted, then from now on, for all calls $\mathcal{G}_{\text{att}}$.`VerifyAttestation`$(sid, \mathsf{prog}_f[\{(P', vk_{P'})\}_{pid_{P'} \in \mathcal{P}}], eid, \mathsf{aid}[P, eid], (P, X', \mathsf{Y}[P, eid], \mathsf{s}[P, eid]))$ from P, do not forward and return 1 in the name of $\mathcal{G}_{\text{att}}$.
  - Await ("`identify`", fid, c') from P.
  - Let H abort if P is corrupted and $\Sigma.\mathsf{Verify}(vk_P, \mathsf{s}[P, eid], \mathsf{AE.Dec}(\mathsf{key}[P, eid], c')) = 0$.
- On call $\mathcal{G}_{\text{att}}$.`Install`(sid, prog') from corrupted party or corrupted host P:
  - If $\mathsf{prog}' = \mathsf{prog}_f[\,\cdot\,]$[7]:
    - Do not forward the call.
    - Send ("`candidateEid`") to $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ and await ("`candidateEid`", eid').
    - Store ("`enclave`", P, eid', prog').

    Else:
    - Forward the `Install` call to $\mathcal{G}_{\text{att}}$ and await eid'.

[7] The protocol prog_f is generally parameterized with the participating parties' extended identities and the corresponding verification keys. However, a corrupted host might try to install an enclave before all keys are registered at $\mathcal{G}_{\text{reg}}$. Corrupted parties could later register the keys chosen by the host here. Since the mapping between parties and keys is therefore still unclear at this point, we use $\cdot$ as a placeholder.

    - ∗ Forward all subsequent calls from corrupted parties to $\mathcal{G}_{\text{att}}$ regarding this eid' and all requests for attestation keys from $\mathcal{G}_{\text{att}}$ to the environment regarding this eid'.
  - Send eid' to P in the name of $\mathcal{G}_{\text{att}}$.
- On call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid', (\text{“}\texttt{keyExchange}\text{”}, P', X'))$ from corrupted party or corrupted host P, if there is a stored record $(\text{“}\texttt{enclave}\text{”}, P, eid', \cdot)$:
  - Do not forward the call.
  - Sample $y \xleftarrow{\$} \mathbb{Z}_p$ and $\mathsf{s}[P', eid'] \xleftarrow{\$} \{0,1\}^\kappa$ and set $\mathsf{X}_{\mathsf{encl}}[P', eid'] := X'$ and $\mathsf{Y}[P', eid'] := g^y$.
  - If P' is honest, set $\mathsf{key}[P', eid'] \xleftarrow{\$} \mathcal{K}$. Otherwise, set $\mathsf{key}[P', eid'] := (X')^y$.
  - Send $(P', X', \mathsf{Y}[P', eid'], \mathsf{s}[P', eid'])$ to P in the name of $\mathcal{G}_{\text{att}}$.
- On call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid', (\text{“}\texttt{attest}\text{”}, P'))$ from corrupted party or corrupted host P, if there is a stored record $(\text{“}\texttt{enclave}\text{”}, P, eid', \mathsf{prog}')$:
  - Do not forward the call.
  - Draw $aid_1 \xleftarrow{\$} \{0,1\}^\kappa$. If $\mathsf{key}_{\mathsf{aid}}[P', eid'] = \bot$, ask the adversary for a key and set $\mathsf{key}_{\mathsf{aid}}[P, eid]$ to this key if P is corrupted. If P is honest, set $\mathsf{key}_{\mathsf{aid}}[P', eid'] \leftarrow \Pi.\mathsf{KeyGen}(1^\kappa)$ or, if the environment returned $\bot$ to the request for attestation algorithms, set $\mathsf{key}_{\mathsf{aid}}[P', eid'] = 0^\kappa$.
  - Compute $aid_2 \leftarrow \Pi.\mathsf{Mac}(\mathsf{key}_{\mathsf{aid}}[P', eid'], (\text{“}\texttt{attest}\text{”}, sid, \mathsf{prog}', eid', aid_1, (P', \mathsf{X}_{\mathsf{encl}}[P', eid'], \mathsf{Y}[P', eid'], \mathsf{s}[P', eid'])))$.
  - Send $\mathsf{aid}[P', eid'] := aid_1 || aid_2$ to P in the name of $\mathcal{G}_{\text{att}}$.
  - If P' is corrupted, then for all subsequent calls $\mathcal{G}_{\text{att}}.\texttt{VerifyAttestation}(sid, \mathsf{prog}', eid', \mathsf{aid}[P', eid'], (P', \mathsf{X}_{\mathsf{encl}}[P', eid'], \mathsf{Y}[P', eid'], \mathsf{s}[P', eid']))$ from P', do not forward and return 1 in the name of $\mathcal{G}_{\text{att}}$.
- On call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid', (\text{“}\texttt{identify}\text{”}, P', c'))$ from corrupted party or corrupted host P, if there is a stored record $(\text{“}\texttt{enclave}\text{”}, P, eid', \mathsf{prog}_f[\mathsf{vkmap}])$:
  - If P' is honest, there is a stored record $(\text{“}\texttt{fidEid}\text{”}, fid, P', eid')$ and $c' = \mathsf{c}_{\mathsf{ident}}[fid, P']$, or if P' is corrupted, $(P', vk') \in \mathsf{vkmap}$, and $\Sigma.\mathsf{Verify}(vk', \mathsf{s}[P', eid'], \mathsf{AE.Dec}(\mathsf{key}[P', eid'], c')) = 1$, mark P' as identified and return 1 to P. Otherwise, return 0 to P.
- On notification of input with some fid and with length ℓ from honest party P to $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$, if $\mathcal{S}$ has previously received $(\text{“}\texttt{eid}\text{”}, fid, eid)$ from $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$:
  - Generate $c \leftarrow \mathsf{AE.Enc}(\mathsf{key}[P, eid], 0^\ell)$ and set $\mathsf{c}_{\mathsf{inp}}[fid, P] := \mathsf{c}_{\mathsf{inp}}[fid, P] \cup \{c\}$.
  - Send $(\text{“}\texttt{input}\text{”}, fid, c)$ from P to H.
- On message $(\text{“}\texttt{input}\text{”}, fid, c')$ from party P to honest host H, if $\mathcal{S}$ has previously received $(\text{“}\texttt{eid}\text{”}, fid, eid)$ from $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$:
  - If $c' \in \mathsf{L}[eid]$, let H abort.
  - Set $\mathsf{L}[eid] := \mathsf{L}[eid] \cup \{c'\}$.
  - If P is corrupted and $\mathsf{AE.Dec}(\mathsf{key}[P, eid], c') = \bot$, ignore the message.
  - Acknowledge the notification from $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ about the input and wait for a public delayed output $(fid, eid, \mathsf{prog}_f[\{(P', vk_{P'})\}_{pid_{P'} \in \mathcal{P}}], aid, \nu, \mathsf{out})$ from $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ to P' or for a backdoor message $(\text{“}\texttt{output}\text{”}, fid)$ from $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ to $\mathcal{S}$.

  - In case of a public delayed output, send ("output", fid, aid, ν, out) from H to P'.
- On call $\mathcal{G}_{\text{att}}$.Resume(eid, ("input", P', c')) from corrupted party or corrupted host P to $\mathcal{G}_{\text{att}}$, if there is a stored record ("enclave", P, eid, prog$'$):
  - Do not forward the call.
  - If $\mathcal{S}$ has not previously received ("eid", fid, eid) from sh[$\mathcal{F}_{\text{mf}}$]:
    * If at least one party with a PID from $\mathcal{P}$ is honest, ignore the call. Otherwise:
      · Sample $fid \xleftarrow{\$} \{0,1\}^\kappa$.
      · Send ("chooseEid", fid, eid) to sh[$\mathcal{F}_{\text{mf}}$] and receive ("eid", fid, eid).
  - If $c' \in \mathsf{L}[eid]$, ignore this call and all future $\mathcal{G}_{\text{att}}$.Resume calls regarding this eid.
  - Set $\mathsf{L}[eid] \coloneqq \mathsf{L}[eid] \cup \{c'\}$.
  - If P' is honest and $c' \notin c_{\mathsf{inp}}[fid, P']$, ignore the call.
    Else if P' is corrupted:
    * Set $m \coloneqq \mathsf{AE.Dec}(\mathsf{key}[P', eid], c')$.
    * If $m = \bot$ or, if not all parties are marked as identified, ignore the call.
    * Input (fid, m) to sh[$\mathcal{F}_{\text{mf}}$] in the name of P' and wait for a notification from sh[$\mathcal{F}_{\text{mf}}$] about the input.
  - Acknowledge the notification from sh[$\mathcal{F}_{\text{mf}}$] about the input associated to c' and wait for a public delayed output (fid, eid, prog$'$, aid, ν, out) from sh[$\mathcal{F}_{\text{mf}}$] to P'' or for a backdoor message ("output", fid) from sh[$\mathcal{F}_{\text{mf}}$] to $\mathcal{S}$.
  - In case of a public delayed output, output (ν, P'', out) to H in the name of $\mathcal{G}_{\text{att}}$. Otherwise, output (ν, $\bot$, ε) to H in the name of $\mathcal{G}_{\text{att}}$.
- On call $\mathcal{G}_{\text{att}}$.Resume(eid, "attestOutput") from corrupted party or corrupted host P to $\mathcal{G}_{\text{att}}$, if $\mathcal{S}$ has previously received ("eid", fid, eid) from sh[$\mathcal{F}_{\text{mf}}$]:
  - Do not forward the call.
  - If the last response of $\mathcal{S}$ to a call $\mathcal{G}_{\text{att}}$.Resume(eid, ("input", $\cdot$, $\cdot$)) was of the form ($\cdot$, $\bot$, ε), ignore the call.
  - Send aid to P in the name of $\mathcal{G}_{\text{att}}$, where aid is the attestation ID from the last delayed output (fid, aid, ν, out) from sh[$\mathcal{F}_{\text{mf}}$].
- On message ("output", fid, aid, ν, out$'$) from host H to honest party P:
  - If not $\mathcal{G}_{\text{att}}$.VerifyAttestationAll(sid, prog$_f$[$\{(P', vk_{P'})\}_{pid_{P'} \in \mathcal{P}}$], eid, aid, (ν, P, out$'$)) or if $\nu \in \mathsf{oids}_P$, let P abort.
  - Set $\mathsf{oids}_P \coloneqq \mathsf{oids}_P \cup \{\nu\}$.
  - Allow the delayed output from sh[$\mathcal{F}_{\text{mf}}$] that corresponds to this message.
- On call $\mathcal{G}_{\text{att}}$.VerifyAttestationAll(sid, prog$_f$[$\{(P', vk_{P'})\}_{pid_{P'} \in \mathcal{P}}$], eid, aid, (ν, P', out$'$)) from corrupted party P to $\mathcal{G}_{\text{att}}$, forward the call to $\mathcal{G}_{\text{att}}$ and forward back the result to P.

We show indistinguishability of the following hybrid games G_i. Let out$_i$ denote the output of the environment in hybrid G_i.

G_0: The real execution with $\pi^{\mathsf{pub}}_{\mathsf{mult},f}$, $\mathcal{G}_{\text{att}}$, $\mathcal{G}_{\text{reg}}$, and the dummy adversary.

G_1: Execution with an ideal functionality $\mathcal{F}_1$ and $\mathcal{S}_1$, where $\mathcal{F}_1$ reports all inputs to the simulator $\mathcal{S}_1$ and lets $\mathcal{S}_1$ determine the output. $\mathcal{S}_1$ works by executing the protocol on behalf of all honest parties and makes outputs for the honest parties through $\mathcal{F}_1$.

G_2: Execution with functionality $\mathcal{F}_2$. The simulator $\mathcal{S}_2$ no longer calls $\mathcal{G}_{\text{att}}$.`Install`, $\mathcal{G}_{\text{att}}$.`Resume` and $\mathcal{G}_{\text{att}}$.`VerifyAttestation` for honest parties. Instead, it performs the computation itself and keeps the state $\mathcal{G}_{\text{att}}$ would otherwise keep between these calls. Instead of $\mathcal{G}_{\text{att}}$.`Resume`$(eid,$"`attestOutput`") calls by honest parties, $\mathcal{S}_2$ calls $\mathcal{G}_{\text{att}}$.`AddAttestationAll` via $\mathcal{F}_2$ for the corresponding output it computed. For corrupted parties, $\mathcal{S}_2$ no longer forwards the $\mathcal{G}_{\text{att}}$.`Install` calls if the program is of the correct form, $\mathcal{G}_{\text{att}}$.`Resume` calls if there is a corresponding "`enclave`" record, and $\mathcal{G}_{\text{att}}$.`VerifyAttestation` calls corresponding to $\mathcal{G}_{\text{att}}$.`Install` calls that were not forwarded. Instead, it performs the computations as for honest parties. To determine the enclave ID, $\mathcal{S}_2$ sends backdoor messages to $\mathcal{F}_2$.

G_3: Execution with functionality $\mathcal{F}_3 = \mathcal{F}_2$. The simulator $\mathcal{S}_3$ now uses a random key k for honest parties, instead of the key $\mathsf{key}[P, eid] = g^{xy}$ from the simulated Diffie-Hellman key exchange.

G_4: Execution with functionality $\mathcal{F}_4 = \mathcal{F}_3$. The simulator $\mathcal{S}_4$ now uses a key $k_{\mathsf{aid}} \leftarrow \Pi.\mathsf{KeyGen}(1^\kappa)$ for honest parties, instead of the key $k[P, eid] \leftarrow \Pi.\mathsf{KeyGen}(1^\kappa)$ sampled by $\mathcal{G}_{\text{att}}$ to generate aid_2, if the environment chose an algorithm $\Pi.\mathsf{KeyGen}$. Otherwise, it sets $k_{\mathsf{aid}} = 0^\kappa$.

G_5: Execution with the ideal functionality $\mathcal{F}_5 = \mathcal{F}_4$, but now the simulator $\mathcal{S}_5$ aborts, if it receives a call $\mathcal{G}_{\text{att}}$.`Resume`$(eid, (\text{"}\texttt{input}\text{"}, P, c))$ from H for an honest party P and a ciphertext c, which has not been sent to H in the name of P.

G_6: Execution with functionality $\mathcal{F}_6$, that no longer tells inputs of honest parties to the simulator. $\mathcal{S}_6$ now uses ciphertexts of 0^ℓ, with the respective appropriate length ℓ, whenever it would create ciphertexts of an input or of a signature by an honest party. The simulated enclave does not decrypt the ciphertexts from honest parties and instead uses the inputs received by $\mathcal{F}_6$ and always accepts the identification of honest parties.

G_7: Execution with functionality $\mathcal{F}_7 := \mathsf{sh}[\mathcal{F}_7']$, where $\mathcal{F}_7'$ no longer calls $\mathcal{G}_{\text{att}}$.`AddAttestationAll`. $\mathcal{S}_7$ no longer calls $\mathcal{G}_{\text{att}}$.`VerifyAttestationAll` and $\mathcal{G}_{\text{reg}}$.`Register` in the name of honest parties. Instead, $\mathsf{sh}[\mathcal{F}_7']$ calls $\mathcal{G}_{\text{att}}$.`AddAttestationAll` and $\mathcal{G}_{\text{reg}}$.`Register`. $\mathsf{sh}[\mathcal{F}_7']$ now also responds to the backdoor messages regarding the enclave ID.

G_8: The ideal execution with $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$ and simulator $\mathcal{S}$.

Indistinguishability:

Claim 1. out_0 and out_1 are distributed identically.

Proof. The difference between G_0 and G_1 is only syntactical. Therefore, these games behave identically and the outputs are indistinguishable.

Claim 2. out_1 and out_2 are distributed identically.

Proof. As $\mathcal{G}_{\text{att}}$ does not hold any shared state between enclaves with different eid, and because the state changes, that happen in $\mathcal{G}_{\text{att}}$ in G_1 can only be observed by parties in the current session, with exception of the public attestations of the outputs, $\mathcal{F}_2$ can return the correct output corrupted parties receive and honest parties would receive in G_2. The $\mathcal{G}_{\text{att}}$.`AddAttestationAll` call by $\mathcal{F}_2$ ensures that the public attestation of the output is also consistent with G_1.

Claim 3. Assuming the DDH problem is hard, out_2 and out_3 are computationally indistinguishable.

Proof. Let $\mathcal{C}$ be the challenger in the DDH game and $\mathcal{A}$ the adversary. To determine the choice bit b', $\mathcal{A}$ internally runs G_2 and G_3. Assume, $\mathcal{A}$ can distinguish these games with non-negligible probability, then $\mathcal{A}$ has non-negligible probability in winning the DDH game. This is a contradiction to the DDH assumption, which concludes this proof.

Claim 4. out_3 and out_4 are distributed identically.

Proof. k_{aid} is generated with the same algorithm $\Pi.\mathsf{KeyGen}$ as $k[P, eid]$ and therefore follows the same distribution. If the environment did not pick an algorithm Π, the keys would both be set to $k_{\mathsf{aid}} = k[P, eid] = 0^{\kappa}$. Since the first part of the aid, aid_1, is sampled randomly in any case, security is not affected.

Claim 5. Assuming AE is an INT-CTXT secure encryption scheme, out_4 and out_5 are computationally indistinguishable.

Proof. For the simulator to abort in G_5, the adversary $\mathcal{A}$ would have to create ciphertexts, that are both fresh and valid for the key of an honest party. By outputting such ciphertexts, $\mathcal{A}$ would win the INT-CTXT game with non-negligible probability. This is a contradiction to the assumption that AE is an INT-CTXT secure encryption scheme, which concludes this proof.

Claim 6. Assuming AE is an IND-CPA secure encryption scheme, out_5 and out_6 are computationally indistinguishable.

Proof. Let $\mathcal{C}$ be the challenger in the IND-CPA game and $\mathcal{A}$ the adversary. To determine the choice bit b', $\mathcal{A}$ internally runs G_5 and G_6. Assume, $\mathcal{A}$ can distinguish these games with non-negligible probability, then $\mathcal{A}$ has non-negligible probability in winning the IND-CPA game. This is a contradiction to the assumption that AE is an IND-CPA secure encryption scheme, which concludes this proof.

Claim 7. out_6 and out_7 are distributed identically.

Proof. The calls have been moved to the ideal shell, but the calls themselves stay the same, except for $\mathcal{G}_{\text{att}}$.`VerifyAttestationAll`, which has been removed. The $\mathcal{G}_{\text{att}}$.`VerifyAttestationAll` call is redundant in G_6 because the public attestation was created by the functionality itself.

Claim 8. out_7 and out_8 are distributed identically.

Proof. As the evaluation of f in the simulated enclave in $\mathcal{F}_7'$ and the evaluation of f in $\mathcal{F}_{\mathsf{m}f}$ are on the same input and have the same output, and because $\mathcal{S}$ simulates the protocol execution like $\mathcal{F}_7'$, the games behave identically.

Using the above claims, we showed that $\mathsf{G}_0 \approx \mathsf{G}_8$. The theorem follows.

G Proof of Theorem 7 (Special Global Public Verifiability with $\mathcal{G}_{\mathbf{att}}$)

Theorem 7 *Assuming* AE *is an INT-CTXT, IND-CPA secure, perfectly correct encryption scheme and the DDH assumption holds, and that* Σ *is an EUF-CMA secure signature scheme, the protocol* $\pi^{\mathsf{pub}}_{\mathsf{mult},f}$ *is specially publicly verifiable with respect to* $\mathcal{F}_{\mathsf{m}f}$ *using* $\mathcal{G}_{\mathrm{att}}$ *and* $\mathcal{G}_{\mathrm{reg}}$.

To prove Thm. 7 and show that public verifiability according to Def. 6 can be accomplished with our model of $\mathcal{G}_{\mathrm{att}}$ and $\mathcal{F}_{\mathsf{m}f}$ as $\mathcal{F}$, we give the algorithm of the verifier in Def. 10.

Definition 10. *Verifier* $V^{\mathcal{O}}(1^\kappa, t, U, y_i)$
- Upon verification, ask the oracle $\mathcal{O}$ to run $\mathcal{G}_{\mathrm{att}}$.`VerifyAttestationAll`($sid$, $\mathsf{prog}_f[\{(P, vk_P)\}_{pid_P \in \mathcal{P}}]$, eid, aid, y_i), where
  - sid is the SID of IDEAL($\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$) and $\mathsf{prog}_f[\{(P, vk_P)\}_{pid_P \in \mathcal{P}}]$ is the enclave program defined as in Constr. 2. The SID sid and the extended identitities and public keys of the parties that prog_f is parameterized with are part of the trusted information in t.
  - eid and aid are part of the untrusted information in U. If multiple eids are associated with the output value y_i, try all of them and if the oracle responds with 1 at least once, continue with this answer.
- Output the response of the oracle.

$\mathcal{G}_{\mathrm{att}}$.`VerifyAttestationAll`($sid, \mathsf{prog}[\{(P, vk_P)\}_{pid_P \in \mathcal{P}}], eid, aid, y_i$) is going to return 1 if a corresponding entry exists in the verifier list V. Otherwise, it is going to return 0. Entries are added to this list only in the $\mathcal{G}_{\mathrm{att}}$.`AddAttestation` and $\mathcal{G}_{\mathrm{att}}$.`AddAttestationAll` interfaces, which can only be called by enclaves and $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$.

Since in our model, attestations cannot be revoked and all outputs of $\mathcal{F}_{\mathsf{m}f}$ are added to V, with the respective attestations being part of the outputs by $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$, $\Pr[V^{\mathcal{O}}(1^\kappa, t, U, y_i) = 1] \geq 1 - \mathsf{negl}(\kappa)$ is true for all $(\cdot, y_i) \in Y$.

The adversary could only add attestations to V by installing an enclave. However, the enclave program installed has to match the one input into the verification interface for the verification to go through. In that case, the enclave does not actually exist but rather is simulated by the simulator. The publicly verifiable attestations are then created by $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$, which match the outputs of $\mathcal{F}_{\mathsf{m}f}$ by construction. This is still the case even if all parties are corrupted. Thus, the adversary does not have a chance to create attestations that are accepted by the verifier, but do not match any of $\mathcal{F}_{\mathsf{m}f}$'s outputs. Therefore,

$\Pr[V^{\mathcal{O}}(1^\kappa, t, U', y_i') = 0 \mid (U', y_i') \leftarrow \mathcal{B}(1^\kappa, t, U, Y, \text{view}_{\mathcal{Z}}) \wedge ((\cdot, y_i') \notin Y)] \geq 1 - \mathsf{negl}(\kappa)$ is fulfilled as well and together with the proof of Thm. 6 we have shown that $\pi^{\mathsf{pub}}_{\mathsf{mult},f}$ is specially publicly verifiable with respect to $\mathcal{F}_{\mathsf{m}f}$.

H Impossibility of Global Public Verifiability using $\mathcal{G}^{\mathbf{PST}}_{\mathbf{att}}$

In the following, we propose a variant of Def. 6. In Def. 11, ϕ is not restricted to just one of two protocols. It can be any protocol that fulfills the few conditions which are explicitly stated. Additionally, π is allowed to use the global augmented common reference string functionality $\mathcal{G}_{\text{acrs}}$ that was introduced by Canetti, Dodis, Pass and Walfish [Can+07, Fig. 6] and later used by Pass, Shi and Tramèr for their MPC protocol which only relies on a single enclave [PST17, Sect. 2.5]. The CRS stored in $\mathcal{G}_{\text{acrs}}$ is then an additional entry in the t input to V. We call this variant *weak* enclave-based global public verifiability instead of *special* to differentiate between the two definitions.

Definition 11. *Weak Enclave-Based Global Public Verifiability* Let ξ be an identity bound that excludes all extended identities with code $\mathsf{sh}[\mathcal{F}_{\mathsf{m}f}]$. Let $\mathcal{F}$ be an ideal functionality, such that the output values produced by $\mathcal{F}$ only depend on the inputs to $\mathcal{F}$ and the contents of $\mathcal{F}$'s random tape. For $\mathcal{G}^*_{\text{att}} \in \{\mathcal{G}^{\text{PST}}_{\text{att}}, \mathcal{G}^{\text{MAC}}_{\text{att}}, \mathcal{G}_{\text{att}}\}$, let γ^* denote the combination of the ideal protocols for $\mathcal{G}^*_{\text{att}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$. We say that a protocol π is weakly publicly verifiable with respect to $\mathcal{F}$ using $\mathcal{G}^*_{\text{att}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ if

- there is a PPT protocol ϕ such that
  - for each possible session of ϕ, a polynomial-size superset of all extended identities in the extended session can be efficiently derived from the SID, the input sequence and the output sequence of the session,
  - in each possible extended session of ϕ, there is at most one instance of $\mathcal{F}$ or of a structured ITM that contains $\mathcal{F}$ as the body beneath one or multiple layers of shells that do not interfere with $\mathcal{F}$'s internal computations,
  - from each sequence of input values, an input sequence for ϕ can be efficiently derived which causes ϕ to pass the desired input sequence to $\mathcal{F}$,
  - the output values of $\mathcal{F}$ are efficiently and unambiguously extractable from the output sequence of ϕ,

  and there is a PPT verifier V such that
- for every PPT adversary $\mathcal{A}$, there is a PPT simulator $\mathcal{S}$ such that for every PPT ξ-identity-bounded environment $\mathcal{Z}$, it holds that the ensembles $\mathsf{Exec}(\mathsf{M}[\pi, \gamma^*], \mathcal{A}, \mathcal{Z})$ and $\mathsf{Exec}(\mathsf{M}[\phi, \gamma^*], \mathcal{S}, \mathcal{Z})$ are computationally indistinguishable and
- for every PPT adversary $\mathcal{B}$, for every $a \in \{0,1\}^{\leq \mathsf{poly}(\kappa)}$ and for uniformly sampled randomness r, the following holds at the end of $\mathsf{Exec}(\mathsf{M}[\phi, \gamma^*], \mathcal{S}, \mathcal{Z})(\kappa, a; r)$:
  - $\forall(\cdot, y_i) \in Y : \Pr[V^{\mathcal{O}}(1^\kappa, t, U, y_i) = 1] \geq 1 - \mathsf{negl}(\kappa)$,
  - $\Pr[V^{\mathcal{O}}(1^\kappa, t, U', y_i') = 0 \mid (U', y_i') \leftarrow \mathcal{B}(1^\kappa, t, U, Y, \text{view}_{\mathcal{Z}}) \wedge (\cdot, y_i') \notin Y] \geq 1 - \mathsf{negl}(\kappa)$,

where

- $\mathcal{O}$ is an oracle that does nothing if $\mathcal{G}^*_{\text{att}} \in \{\mathcal{G}^{\text{PST}}_{\text{att}}, \mathcal{G}^{\text{MAC}}_{\text{att}}\}$, or it runs $\mathcal{G}^*_{\text{att}}$.`VerifyAttestationAll`, using the last configuration of $\mathcal{G}^*_{\text{att}}$, if $\mathcal{G}^*_{\text{att}} = \mathcal{G}_{\text{att}}$.
- t is a tuple containing the SID of ϕ, the attestation verification key vk if $\mathcal{G}^*_{\text{att}}$ has one, all pairs of extended identities and public keys stored in $\mathcal{G}_{\text{reg}}$ that belong to the session with this SID, and the CRS stored in $\mathcal{G}_{\text{acrs}}$.
- U is the set of all those tuples of enclave outputs, enclave IDs, enclave programs and signatures or attestation IDs which are stored in $\mathcal{G}^*_{\text{att}}$ and also contained in an output of ϕ.
- Y is the set of outputs of $\mathcal{F}$.

Definition 12. *Global Augmented Common Reference String*

$\mathcal{G}_{\text{acrs}}[\mathsf{Setup}, \mathsf{Extract}]$

On initialize: Sample $sk \xleftarrow{\$} \{0,1\}^\kappa$ and compute $pk := \mathsf{Setup}(sk)$.
On input `Crs`() from some party P: Send a public delayed output pk to P.
On input `Idk`() from a corrupted party P: Output
$sk_P := \mathsf{Extract}(pk, pid_P, sk)$ to P.

We will show that even this weaker variant of enclave-based public verifiability is not achievable for certain functionalities in the model by Pass, Shi and Tramèr. For this purpose, we define the following functionality:

Definition 13. *Randomness Generator Functionality* $\mathcal{F}_{\text{mrand}}$ Parameterized with a party P. On input fid from P, sample $r \xleftarrow{\$} \{0,1\}^\kappa$ and generate a public delayed output (fid, r) to P. Ignore all further inputs with the same FID.

Theorem 9. *There is no PPT protocol π that is weakly publicly verifiable with respect to $\mathcal{F}_{\text{mrand}}$ using $\mathcal{G}^{\text{PST}}_{\text{att}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$.*

Proof. Assume that there is a PPT protocol π which is weakly publicly verifiable with respect to $\mathcal{F}_{\text{mrand}}$ using $\mathcal{G}^{\text{PST}}_{\text{att}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$. This means on the one hand that there is a modified ideal protocol ϕ for $\mathcal{F}_{\text{mrand}}$ and a PPT simulator $\mathcal{S}$ such that no PPT environment can distinguish between an execution of π in the presence of $\mathcal{G}^{\text{PST}}_{\text{att}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ with the dummy adversary and an execution of ϕ in the presence of $\mathcal{G}^{\text{PST}}_{\text{att}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ with adversary $\mathcal{S}$. Consider an environment $\mathcal{Z}_1$ in the latter case that passes inputs to ϕ with arbitrary SID and arbitrary FID, such that $\mathcal{F}_{\text{mrand}}$ will receive input fid. $\mathcal{F}_{\text{mrand}}$ then outputs (fid, r) with a random string r. On the other hand, the fact that π is weakly publicly verifiable with respect to $\mathcal{F}_{\text{mrand}}$ means that, for t and U constructed like in Def. 11 with respect to the execution of ϕ, $V(1^\kappa, t, U, r)$ outputs 1 with overwhelming probability.

Now consider an execution of π in the presence of $\mathcal{G}^{\text{PST}}_{\text{att}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ with the dummy adversary and an environment $\mathcal{Z}_2$ that behaves as follows: At the

beginning, $\mathcal{Z}_2$ instructs the adversary to corrupt all parties. Then $\mathcal{Z}_2$ simulates an execution of ϕ in the presence of $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ with adversary $\mathcal{S}$ and environment $\mathcal{Z}_1$. However, $\mathcal{Z}_2$ does not simulate $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{reg}}$ or $\mathcal{G}_{\text{acrs}}$. Whenever a virtual ITI sends an input to $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{reg}}$ or $\mathcal{G}_{\text{acrs}}$, $\mathcal{Z}_2$ forwards the input to the external adversary with the instruction to forward the input to the corresponding external global functionality in the name of the corresponding corrupted ITI. The response is then forwarded back to the virtual ITI. Note that there cannot be an ITI in the extended session of ϕ that does not have a counterpart in π and registers a value at $\mathcal{G}_{\text{reg}}$, since otherwise, π and ϕ would be distinguishable by retrieving that value. When the simulation terminates, $\mathcal{Z}_2$ constructs t and U and outputs the result of $V(1^\kappa, t, U, r)$. $\mathcal{Z}_2$ is able to construct t because it can query $\mathcal{G}_{\text{att}}^{\text{PST}}.\texttt{getpk}()$, $\mathcal{G}_{\text{reg}}.\texttt{Retrieve}(P)$ and $\mathcal{G}_{\text{acrs}}.\texttt{Crs}()$ directly for all ITIs P in the extended session of π and it can construct U because all enclave outputs $\mathcal{G}_{\text{att}}^{\text{PST}}$ generated were forwarded to $\mathcal{Z}_2$. If any of the responses to the queries were suppressed, $\mathcal{Z}_2$ would output 0. In this execution, this cannot happen because $\mathcal{Z}_2$ is in control of the dummy adversary, which is the only entity that would be able to suppress the outputs. The output of V does not depend on whether the values t and U and the output (fid, r) were computed in an actual execution of ϕ in the presence of $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ with adversary $\mathcal{S}$ and an environment $\mathcal{Z}_1$ or whether they were generated by the simulation inside $\mathcal{Z}_2$ that interacts with external $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ functionalities via corrupted parties. This means that V and therefore $\mathcal{Z}_2$ outputs 1 with overwhelming probability.

The fact that r is sampled independently from the rest of the execution means that it does not make a difference whether r is sampled while the execution is running or before the execution started. Consequently, there is a set R of fixed values for r such that with any of them, $\mathcal{Z}_2$ outputs 1 with overwhelming probability. The ratio of $|R|$ to $|\{0,1\}^\kappa|$ must be overwhelming, since otherwise $\mathcal{F}_{\text{mrand}}$ would have non-negligible probability to sample a value from $\{0,1\}^\kappa \setminus R$, making $\mathcal{Z}_2$ output 0 with non-negligible probability. Now we can choose, for example, the value $\hat{r} \in R$ with the lowest value when interpreted as a binary number. Due to the size of R, the probability that a uniformly sampled $r \xleftarrow{\$} \{0,1\}^\kappa$ is less or equal to such a low value is negligible.

Finally, consider an execution of ϕ in the presence of $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ with adversary $\mathcal{S}$ and environment $\mathcal{Z}_3$. $\mathcal{Z}_3$ is identical to $\mathcal{Z}_2$, except that the virtual $\mathcal{F}_{\text{mrand}}$ in $\mathcal{Z}_3$'s simulation always outputs $\hat{r}$. $\mathcal{Z}_3$ outputs 1 with overwhelming probability, like $\mathcal{Z}_2$, since otherwise $\mathcal{Z}_3$ would have a non-negligible advantage in distinguishing the real world with protocol π and the ideal world with protocol ϕ. Note that $\mathcal{Z}_3$ constructs t accurately with overwhelming probability because it queries $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$ directly, because only ITIs of ϕ that have a counterpart in π register with $\mathcal{G}_{\text{reg}}$ and because $\mathcal{S}$ cannot suppress any outputs of $\mathcal{G}_{\text{reg}}$ to $\mathcal{Z}_3$ without enabling $\mathcal{Z}_3$ to distinguish between the real and the ideal world. Let U' be the value that $\mathcal{Z}_3$ constructs in place of U. That $\mathcal{Z}_3$ is able to construct the value means that a PPT adversary $\mathcal{B}$ with access to the view of $\mathcal{Z}_3$ is also able to construct U'. Lastly, note that the external $\mathcal{F}_{\text{mrand}}$ samples randomness uniformly, so the probability that it outputs $\hat{r}$ is negligible.

Therefore, $V(1^\kappa, t, U', \hat{r})$ outputs 0 with overwhelming probability. However, this is a contradiction to the fact that $\mathcal{Z}_3$ outputs 1 with overwhelming probability. We conclude that our assumption, that there is a PPT protocol π which is weakly publicly verifiable with respect to $\mathcal{F}_{\text{mrand}}$ using $\mathcal{G}_{\text{att}}^{\text{PST}}$, $\mathcal{G}_{\text{reg}}$ and $\mathcal{G}_{\text{acrs}}$, is false.

In contrast, Theorem 7 shows that our $\mathcal{G}_{\text{att}}$ functionality can be used to achieve public verifiability with respect to $\mathcal{F}_{\text{mrand}}$, since $\mathcal{F}_{\text{mrand}}$ is equivalent to $\mathcal{F}_{\mathsf{m}f}$ parameterized with the following function:

$$f(\text{state}, r, p, v) = \begin{cases} (1, P, r') & \text{state} = \varepsilon \wedge p = P \wedge v = \varepsilon \\ & \quad \wedge\ r' \text{ is } \kappa\text{-bit prefix of } r \\ (\text{state}, \bot, \varepsilon) & \text{otherwise} \end{cases}$$

I Proof of Theorem 8 (Realizing $\mathcal{G}_{\text{att}}$ Using $\mathcal{G}_{\text{att}}^{\text{MAC}}$)

In this section we provide detailed descriptions of the wrapper $\mathcal{W}[\cdot]$ (Fig. 7) and the shell W (Fig. 8) as well as the security proof of Thm. 8.

I.1 Wrapper

The wrapper bridges the gap between $\mathcal{G}_{\text{att}}$ and $\mathcal{G}_{\text{att}}^{\text{MAC}}$, covering interfaces that $\mathcal{G}_{\text{att}}^{\text{MAC}}$ has but $\mathcal{G}_{\text{att}}$ does not. It does so by fully shielding $\mathcal{G}_{\text{att}}^{\text{MAC}}$ and by processing any call to $\mathcal{G}_{\text{att}}^{\text{MAC}}$ before passing it on to $\mathcal{G}_{\text{att}}^{\text{MAC}}$. Each party p runs $\mathcal{W}$, *i.e.*, $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W, p]$, where W is the shell program we present in Fig. 8 and explain below. $\mathcal{W}$ takes calls from an outer protocol that would normally call $\mathcal{G}_{\text{att}}$. It parses these calls, either forwarding them to $\mathcal{G}_{\text{att}}^{\text{MAC}}$ or performing additional steps to bridge the differences between the functionalities. First, $\mathcal{G}_{\text{att}}^{\text{MAC}}$ requires parties to register as verifiers. In $\mathcal{G}_{\text{att}}$, an enclave host calls `AddAttestation`, which outputs an attestation for a designated verifier. Before adding the attestation, the enclave host's wrapper has to take care of registering the verifier. Upon receiving a message to register a party P_j from $\mathcal{G}_{\text{att}}^{\text{MAC}}$, it performs the registration. If P_j does not host an enclave, $\mathcal{W}$ has to register P_j at $\mathcal{G}_{\text{att}}^{\text{MAC}}$ and the host's party requests a key share from P_j which in turn can be forwarded to $\mathcal{G}_{\text{att}}^{\text{MAC}}$. If P_j hosts an enclave, $\mathcal{W}$ registers P_j directly at $\mathcal{G}_{\text{att}}^{\text{MAC}}$ via the `Register` interface. The obtained symmetric key is then stored in the wrapper of P_j as well as inside the enclave in which registration took place. Once registration is completed, $\mathcal{W}$ resumes the enclave program with a special input provided by the shell to continue the attestation process.

I.2 Shell

The shell program intercepts the installed program and adds designated verifier attestations at $\mathcal{G}_{\text{att}}^{\text{MAC}}$ via the registration interface. Any program prog can run inside this shell. $\mathcal{W}$ is responsible for installing the shelled program $W[\mathsf{prog}]$ in place of prog to enable registration. In Fig. 8, we define the shell

in a similar fashion as enclave programs. An enclave program works by receiving inputs from $\mathcal{G}_{\text{att}}^{\text{MAC}}$ which then the enclave host prompts by calling $\mathcal{G}_{\text{att}}^{\text{MAC}}$.`Resume`$(\cdot, \mathsf{inp})$. The program is executed on inp and is able to call the enclave-internal operations of $\mathcal{G}_{\text{att}}^{\text{MAC}}$, `AddAttestation` and `AddAttestationAll`. Whenever the program prog attempts to call $\mathcal{G}_{\text{att}}^{\text{MAC}}$.`AddAttestation`, the shell intercepts and returns ((“`register`”, $\cdot$), $\cdot$) to $\mathcal{W}$. This starts the registration process between a wrapper and $\mathcal{G}_{\text{att}}^{\text{MAC}}$ as explained above. Upon completion, $\mathcal{W}$ resumes the enclave program by using the shell-specific input “`continueAttest`”, $\mathcal{G}_{\text{att}}^{\text{MAC}}$.`resume`(“`continueAttest`”, $\cdot$). Any other call to $\mathcal{G}_{\text{att}}^{\text{MAC}}$ inside prog is simply forwarded to $\mathcal{G}_{\text{att}}^{\text{MAC}}$. Outputs that prog returns are forwarded by $W[\mathsf{prog}]$. To this end, the remainder of prog remains unchanged.

For technical details on the wrapper and shell, we refer to [MK25].

Theorem 8 *Assuming $\Pi = (\mathsf{KeyGen}, \mathsf{Mac}, \mathsf{Verify})$ is an sEUF-CMA secure MAC scheme and $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is an sEUF-CMA signature schemes, and the DDH assumption holds, $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\,\cdot\,], p]$ UC-realizes $\mathcal{G}_{\text{att}}$ in the presence of the global functionality $\mathcal{G}_{\text{att}}^{\text{MAC}}$ under static corruption.*

To prove the theorem, we first define the simulator. This simulator covers the cases where all parties are honest, only parties without enclaves are corrupted, only parties with enclaves are corrupted and a combination of enclave hosts and parties without enclaves are corrupted. We do not make assumptions about the amount of corrupted parties, *i.e.*, all parties might be corrupted. In general, the simulator does not let parties install an enclave on $\mathcal{G}_{\text{att}}^{\text{MAC}}$, instead it installs enclaves on $\mathcal{G}_{\text{att}}$. Any operation that cannot be performed by the simulator is forwarded to $\mathcal{G}_{\text{att}}$. Operations that are not supported by $\mathcal{G}_{\text{att}}$, such as registration of designated verifiers, are performed by the simulator.

As explained in Sect. 3 the simulator has the ability to upload attestation algorithms and a key to $\mathcal{G}_{\text{att}}$. This allows the simulator to let $\mathcal{G}_{\text{att}}$ create attestations on values unknown to the simulator while corrupted parties holding the respective key can verify these attestations locally. This is necessary since corrupted parties expect to perform a key exchange and verify attestations using a verification key. To prevent distinguishability by the environment by checking MACs or signatures in the ideal world, in $\mathcal{G}_{\text{att}}$, the MAC resp. signature must contain the wrapped enclave program $W[\mathsf{prog}]$ instead of prog. The attestation algorithms used by $\mathcal{G}_{\text{att}}^{\text{MAC}}$ thus need to be slightly adapted by the simulator before being sent to $\mathcal{G}_{\text{att}}$. The signing key to create publicly verifiable attestations is chosen by the simulator to perform a plausible key exchange with corrupted parties in the name of $\mathcal{G}_{\text{att}}^{\text{MAC}}$.

More specifically, the simulator works by receiving inputs from the adversary and either the call to $\mathcal{G}_{\text{att}}$ with modified inputs (in the case of `Install` or `Resume` messages), or in the case of inputs related to the registration, performing the necessary steps to establish a shared key with the adversary. This registration procedure is triggered by $\mathcal{G}_{\text{att}}$ when an enclave calls `AddAttestation`. The simulator is required to provide a key that $\mathcal{G}_{\text{att}}$ uses to create a designated-verifier attestation. To answer this request, the simulator acts as $\mathcal{G}_{\text{att}}^{\text{MAC}}$ and prompts the

adversary with the respective messages to establish this key. Once this procedure has succeeded, the key is returned to $\mathcal{G}_{\text{att}}$. Since the wrapper protocol remembers which party already has a MAC key, $\mathcal{G}_{\text{att}}$ stores the keys as well to prevent further leakage about adding attestations.

Proof. **Definition 14.** *Simulator* $\mathcal{S}$

Let $\mathsf{key} = \emptyset$. Let $\mathsf{Q} = \emptyset$.

- On initialize from $\mathcal{A}$, initialize $\mathcal{G}_{\text{att}}$.
- On message `Initialize` from $\mathcal{G}_{\text{att}}$, let $(sk, vk) \xleftarrow{\$} \Sigma.\mathsf{KeyGen}$ and let $\mathsf{Mac}' := \Pi.\mathsf{Mac}(\cdot, (\cdot, \cdot, W[\cdot], \cdot, \cdot, \cdot)), \mathsf{Sign}' := \Sigma.\mathsf{Sign}(\cdot, (\cdot, \cdot, W[\cdot], \cdot, \cdot, \cdot))$, and return $\Pi' := (\Pi.\mathsf{KeyGen}, \mathsf{Mac}')$, $\Sigma' = (\Sigma.\mathsf{KeyGen}, \mathsf{Sign}', \Sigma.\mathsf{Verify})$, and sk to $\mathcal{G}_{\text{att}}$.
- On input $\texttt{Install}(idx, \widetilde{\mathsf{prog}})$ from $\mathcal{A}$, where $\widetilde{\mathsf{prog}}$ is $W[\mathsf{prog}]$, call $\mathcal{G}_{\text{att}}.\texttt{Install}(idx, \mathsf{prog})$, store and return eid.
- On input `Getvk`() from $\mathcal{A}$, return vk.
- On input $\texttt{Resume}(eid, \mathsf{inp})$ from $\mathcal{A}$, call $\mathcal{G}_{\text{att}}.\texttt{Resume}(eid, \mathsf{inp})$. Return out from $\mathcal{G}_{\text{att}}$ to $\mathcal{A}$.
- On receiving (eid, P_i) and key request for P_j from $\mathcal{G}_{\text{att}}$:
  - If $\mathsf{key}[P_j, eid] \neq \bot$, return $key[P_j, eid]$.
  - If $\mathsf{key}[P_j, eid] = \bot$,
    - If P_i is corrupted, send ("`Register`", P_j) to $\mathcal{A}$ in the name of $\mathcal{G}_{\text{att}}^{\text{MAC}}$.
    - If P_i is not corrupted,
      - If P_j is honest, simulate key exchange between host and P_j and send $\mathsf{key}[eid, P_j]$ to $\mathcal{G}_{\text{att}}$.
      - If P_j is corrupt, send ("`Register`", eid, P_j) to $\mathcal{A}$ in the name of honest party P_i.
- On message ("`Register`", eid, P_j) from $\mathcal{A}$ to honest party P_j:
  - Draw and store $a \leftarrow \mathbb{Z}_p$ in the name of P_j.
  - Send (eid, g^a) to $\mathcal{A}$ in the name of P_j.
- On message $\mathcal{G}_{\text{att}}^{\text{MAC}}.\texttt{Register}(eid, P_j, m)$ or (eid, m) from $\mathcal{A}$:
  - Draw $b \xleftarrow{\$} \mathbb{Z}_p$,
  - Calculate (g^b, σ) as defined in $\mathcal{G}_{\text{att}}^{\text{MAC}}$.
  - $\mathsf{Q} := \mathsf{Q} \cup \{\sigma\}$.
  - Store $\mathsf{key}[eid, P_j] := m^b$.
  - Return (eid, g^b, σ) to $\mathcal{A}$.
  - Send $\mathsf{key}[P_j, eid]$ to $\mathcal{G}_{\text{att}}$.
- On message (eid, g^b, σ) from $\mathcal{A}$ to honest party P_i:
  - If $\sigma \notin \mathsf{Q}$, abort.
  - If σ is valid with respect to vk and $m' = g^a$ for previously stored a, return ("`registered`", eid) to $\mathcal{A}$.
  - Otherwise, abort.
- On message $\mathcal{G}_{\text{att}}^{\text{MAC}}.\texttt{Resume}(\text{"}\texttt{inp}\text{"}, P_j)$ from $\mathcal{A}$, where $\texttt{inp} = \texttt{continue attest}$, return previously stored aid, otherwise forward call to $\mathcal{G}_{\text{att}}$.

We show the indistinguishability of the following hybrids. Let out_i denote the output of the environment in hybrid G_i.

G_0: Real execution with $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], \cdot]$, $\mathcal{G}_{\text{att}}^{\text{MAC}}$ and the dummy adversary.
G_1: Execution of ideal functionality $\mathcal{F}_1$ and $\mathcal{S}_1$, where $\mathcal{F}_1$ reports all inputs to the simulator $\mathcal{S}_1$ and lets $\mathcal{S}_1$ determine the output. $\mathcal{S}_1$ works by receiving messages from the dummy adversary through the environment for p and executes $\mathcal{W}[\mathcal{G}_{\text{att}}^{\text{MAC}}, W[\cdot], \cdot]$ for the honest parties and makes outputs for the honest parties through $\mathcal{G}_1$.
G_2: Execution with ideal functionality $\mathcal{F}_2$, that takes inputs from any party and executes prog on its inputs and gives outputs based on the output of prog. It no longer allows the simulator to determine the output of honest parties. The simulator now uses the keys in the state of the enclave to extract the inputs of corrupted parties and send outputs to corrupted parties from the enclave based on the output of $\mathcal{F}_2$. Note that $\mathcal{F}_2$ performs $\texttt{VerifyAttestation}(idx, \mathsf{prog}, eid, aid, \mathsf{out})$ and $\texttt{VerifyAttestationAll}(idx, \mathsf{prog}, eid, aid, \mathsf{out})$ exactly as $\mathcal{S}_2$ will do by verifying the MAC tag pair (t, m) and the signature pair (σ, m). $\mathcal{F}_2$ aborts if t resp. σ is not valid or (t, m) resp. (σ, m) have not been output previously by $\mathcal{S}_2$.
G_3: Execution with the ideal functionality $\mathcal{F}_3 = \mathcal{F}_2$. For any honest party P_j, the simulator $\mathcal{S}_3$ now uses a random key k, instead of the key $k = g^{ab}$ from the Diffie-Hellman key exchange.
G_4: Execution with an ideal functionality $\mathcal{F}_4 = \mathcal{F}_3$ and the simulator $\mathcal{S}_4 = \mathcal{S}_3$, except that $\mathcal{F}_4$ now executes $\texttt{VerifyAttestation}(idx, \mathsf{prog}, eid, aid, \mathsf{out})$ as specified in $\mathcal{G}_{\text{att}}$.
G_5: Execution with an ideal functionality $\mathcal{F}_5 = \mathcal{F}_4$ and the simulator $\mathcal{S}_5 = \mathcal{S}_4$, except that $\mathcal{F}_5$ now executes $\texttt{VerifyAttestationAll}(idx, \mathsf{prog}, eid, aid, \mathsf{out})$ as specified in $\mathcal{G}_{\text{att}}$.
G_6: Execution with an ideal functionality $\mathcal{F}_6 = \mathcal{F}_5$. For any honest party the simulator $\mathcal{S}_6$ now uses the key k_P from the Diffie-Hellman key exchange again.
G_7: Execution with ideal functionality $\mathcal{G}_{\text{att}}$ and the simulator $\mathcal{S}$.
Let out_i denote the output of the environment in game G_i.

Claim 1. out_0 and out_1 are distributed identically.

Proof. This claim follows directly from the definition of the hybrid games.

Claim 2. out_1 and out_2 are distributed identically.

Proof. This claim follows directly from the definition of the hybrid games.

Claim 3. Assuming the DDH problem is hard, out_2 and out_3 are computationally indistinguishable.

Proof. Let $\mathcal{C}$ be the challenger in the DDH game and $\mathcal{A}$ the adversary. To determine the choice bit b', $\mathcal{A}$ internally runs G_3 and G_4. Assume, $\mathcal{A}$ can distinguish these games with non-negligible probability, then $\mathcal{A}$ has non-negligible probability in winning the DDH game. This is a contradiction to the DDH assumption, which concludes this proof.

Claim 4. Assuming Π is an sEUF-CMA secure MAC scheme, G_3 aborts with negligible probability and out_3 and out_4 are computationally indistinguishable.

Proof. Let $\mathcal{C}$ be the challenger in the sEUF-CMA game for Π and $\mathcal{A}$ the adversary. Assume that G_3 aborts with non-negligible probability, *i.e.*, that the simulator receives a pair (t, m) where t is a valid tag for m, but m has not been previously output by $\mathcal{G}_{\text{att}}^{\text{MAC}}$. Then, $\mathcal{A}$ can win the sEUF-CMA game with non-negligible probability by internally running G_4. This concludes the claim.

Claim 5. Assuming Σ is an sEUF-CMA secure signature scheme, G_4 aborts with negligible probability and out_4 and out_5 are computationally indistinguishable.

Proof. Analogous to proof of Claim 4.

Claim 6. Assuming the DDH problem is hard, out_5 and out_6 are computationally indistinguishable.

Proof. Analogous to proof of Claim 3.

Claim 7. out_6 and out_7 are distributed identically.

Proof. This claim follows directly from the definition of the hybrid games.

Using the above claims, we showed that $\mathsf{G}_0 \approx \mathsf{G}_7$. The theorem follows.