Secure Multiparty Computation from SGX

Abstract. In this paper we show how Isolated Execution Environments (IEE) offered by novel commodity hardware such as Intel's SGX provide a new path to constructing general secure multiparty computation (MPC) protocols. Our protocol is intuitive and elegant: it uses code within an IEE to play the role of a trusted third party (TTP), and the attestation guarantees of SGX to bootstrap secure communications between participants and the TTP. The load of communications and computations on participants only depends on the size of each party's inputs and outputs and is thus small and independent from the intricacies of the functionality to be computed. The remaining computational load– essentially that of computing the functionality – is moved to an untrusted party running an IEE-enabled machine, an attractive feature for Cloud-based scenarios.
Our rigorous modular security analysis relies on the novel notion of labeled attested computation which we put forth in this paper. This notion is a convenient abstraction of the kind of attestation guarantees one can obtain from trusted hardware in multi-user scenarios.
Finally, we present an extensive experimental evaluation of our solution on SGX-enabled hardware. Our implementation is open-source and it is functionality agnostic: it can be used to securely outsource to the Cloud arbitrary off-the-shelf collaborative software, such as the one employed on financial data applications, enabling secure collaborative execution over private inputs provided by multiple parties.

1 Introduction

Secure multiparty computation (MPC) allows a set of mutually distrusting parties to collaboratively carry out a computation that involves their private inputs. The security guarantee that parties get are essentially those provided by carrying out the same computation using a Trusted Third Party (TTP). The computations to be carried out range from simple functionalities, for example where a party commits to a secret value and later on reveals it; or they can be highly complex, for example running sealed bid auctions [11] or bank customer benchmarking [20]. Most of the existent approaches are software only. The trust barrier between parties is overcome using cryptographic techniques that permit computing over encrypted and/or secret-shared data [35,28,19]. Another approach first studied by Katz [31] formalizes a trusted hardware assumption—where users have access to tamper-proof tokens on which they can load arbitrary code—that is sufficient to bootstrap universally composable MPC.

Broadly speaking, this work fits within the same category as that by Katz [15]. However, our starting point is a novel real-world form of trusted hardware that is currently shipped on commodity PCs: Intel's Software Guard Extensions [30]. Our goal is to leverage this hardware to significantly reduce the computational costs of practical secure computation protocols. The main security capability that such hardware offers are Isolated Execution Environments (IEE) – a powerful tool for boosting trust in remote systems under the total or partial control of malicious parties (hijacked boot, corrupt OS, running malicious software, or simply a dishonest service provider). Specifically, code loaded in an IEE is executed in isolation from other software present in the system,[1] and built-in cryptographic attestation mechanisms guarantee the integrity of the code and its I/O behaviour to a remote user.

PROTOCOL OUTLINE. The functionality outlined above suggests a simple and natural design for *general multiparty computation*: load the functionality to be computed into an IEE (which plays the role of a TTP) and have users provide inputs and receive outputs via secure channels to the IEE. Attestation ensures the authenticity of the computed function, inputs and outputs. The resulting protocol is extremely efficient when compared to existing solutions that do not rely on hardware assumptions. Indeed, the load of communications and computations on protocol participants is small and independent of the intricacies of the functionality that is being computed; it depends only on the size of each party's inputs and outputs. The remaining computational load — essentially that of computing the functionality expressed as a transition function in a standard programming language — is moved to an untrusted party running an IEE-enabled machine. This makes the protocol attractive for Cloud scenarios. Furthermore, the protocol is non-interactive in the sense that each user can perform an initial set-up, and then provide its inputs and receive outputs independently of other protocol participants, which means that it provides a solution for "secure computation on the web" [27] with standard MPC security.

Due to its obvious simplicity, variations of the overall idea have been proposed in several practice-oriented works [40,26]. However, currently there is no thorough and rigorous analysis of the security guarantees provided by this solution in the sense of a general approach to MPC. The intuitive appeal of the protocol obscures multiple obstacles in obtaining a formal security proof, including: i. the lack of private channels between the users and the remote machine; ii. the need to authenticate/agree on a computation in a setting where communication between parties is inherently asynchronous and only mediated by the IEE; iii. the need to ensure that the "right" parties are engaged in the computation; iv. dealing with the interaction between different parts of the code that coexist within the same IEE, sharing the same memory space, each potentially corresponding to different users; and v. ensuring that the code running inside an IEE does not leak sensitive information to untrusted code running outside.

In this paper we fill this gap through the following contributions: i. a rigorous specification of the protocol for general MPC computation outlined above; ii. formal security definitions for the security of the overall protocol and that of its components;[2] iii. a modular security analysis of our protocol that relies on a novel notion of labelled attested computation; and iv. an open-source implementation of our protocol and a detailed experimental analysis in SGX-enabled hardware. We give an overview of our results next.

LABELED ATTESTED COMPUTATION. Our protocol relies on ideal functionalities viewed as programs written as transition functions in a programming language compatible with the IEE-enabled machine. We instrument these programs to run inside an IEE and add

[1] We discuss the issue of side-channels that may disrupt the isolation barrier later in the paper.

[2] Since our emphasis is on efficiency and analysing SGX-based protocols used in practice, we do not consider Universal Composability, but rather a simulation-based security model akin to those used for other practical secure computation protocols, e.g. [7].

bootstrapping code that permits protocol participants to establish independent secure channels with the functionality, so that they can provide inputs and receive outputs. The crux of the protocol is a means to provide attestation guarantees which ensures that parties are involved in the “right" run of the protocol (i.e. with the right parties all interacting with the same IEE). We take inspiration from the recent work of Barbosa et al.[3] who provide a formalization for the notion of *attested computation* that can convince a party that its local view of the interaction with a remote IEE matches what actually occurred remotely. This guarantee is close to the one that we need, but it is unfortunately insufficient. The problem is that attested computation a la [3] is concerned with the interaction between a single party and an IEE, and it is non-trivial to extend these guarantees to the interaction of multiple parties with the same IEE when the goal is to reason about *concurrent asynchronous interactions*.

To overcome these problem, we introduce the notion of *labelled attested computation* (LAC), a powerful and clean generalization of the attested computation notion in [3]. In a nutshell, this notion assumes that (parts of) the code loaded in an IEE is marked with labels pertaining to users, and that individual users can get attestation guarantees for those parts of the code that corresponds to specific labels. The gain is that users can now be oblivious of other user's interactions with the IEE, which leads to significantly more simple and efficient protocols. Nonetheless, the user can still derive attestation guarantees about the overall execution of the system, since LAC binds each users' local view to the *same code* running within the IEE, and one can use standard cryptographic techniques to leverage this binding in order to obtain *indirect* attestation guarantees as to the honest executions of the interactions with other users.

We provide syntax and a formal security model for LAC and show how this primitive can be used to deploy arbitrary (labelled) programs to remote IEEs with flexible attestation guarantees. Our provably secure LAC protocol relies on hardware equipped with SGX-like IEEs. Our construction of the MPC protocol then builds on LACs in a modular way. First, we show how to use labelled attested computation schemes[3] to bootstrap an arbitrary number of independent secure channels between local users and an IEE with joint attestation guarantees. We formalize this result as an *utility theorem.* The security of the overall MPC protocol which uses these channels for communication with functionality code inside an IEE is then built on this utility theorem.

IMPLEMENTATION AND EXPERIMENTAL VALIDATION. We conclude the paper with an experimental evaluation of our protocol via a detailed comparison of our solution to state-of-the-art multiparty computation. The experimental results confirm the theoretical performance advantages that we have highlighted above in comparison to non hardware-based solutions. Our implementation of a generic MPC protocol —sgx-mpc— relies on the NaCl[4] cryptographic library [9] and inherits its careful approach to dealing with timing side-channels. We discuss side-channels in SGX-like systems and explain

[3] We use schemes which satisfy the additional notion of *minimal leakage* which ensures that the outsourced instrumented program P^* reveals no information about its internal state beyond what the normal input/output behavior of the original program P would reveal.

[4] `https://nacl.cr.yp.to`

how our *constant-time* code thwarts *all* leaks based on control-flow or memory access patterns that depend on secret data.

Our implementation is functionality agnostic and can be used to outsource to the Cloud arbitrary off-the-shelf collaborative software, enabling multiple parties to jointly execute complex interactive computations without revealing their own inputs. Taking the financial sector as an example, our implementation permits carrying out financial benchmarking [20] using off-the-shelf software, rather than requiring the conversion of the underlying computation into circuit form, as is the case in state-of-the-art secure multiparty computation protocols. One should of course mention that, in order to meet the level of side-channel attack resilience of sgx-mpc, the code that is outsourced to the Cloud should itself be implemented according to the constant-time coding policy. This, however, is a software engineering issue that is outside of the scope of this paper.

RELATED WORK. A relevant line of research leverages trusted hardware to bootstrap entire platforms for secure software execution (e.g. Flicker [36], Trusted Virtual Domains [16], Haven [5]). These are large systems that are currently outside the scope of provable-security techniques. Smaller protocols which solve specific problems are more susceptible to rigorous analysis. Examples of these are secure disk encryption [37], one-time password authentication [29] outsourced Map-Reduce computations [40], Secure Virtual Disk Images [24], two-party computation [26], secure embedded devices [38,33]. Although some of these protocols (e.g., those of Hoekstra et al. [29] and Gupta et al [26]) come only with intuition regarding their security, others—most notably those by Schuster et. al [40]—come with a proof of security. The use of attestation in those protocols is akin to our use of attestation in our general MPC protocol. Provable security of realistic protocols that use trusted hardware-based protocols based on the Trusted Platform Module (TPM) have been considered in [13,41,12,23,22]. The weaker capabilities offered by the TPM makes them more suitable for static attestation than for a dynamic setting like the one we consider in this paper.

In recent independent work Pass, Shi and Tramer [39] formalize attestation guarantees offered by trusted hardware in the Universal Composability setting, and consider the feasibility of achieving UC-secure MPC from such assumptions. Interestingly, they show that in the setting that they consider (UC with a Global Setup (GUC) [14]) multiparty computation is impossible to achieve without additional assumptions, unless *all* parties have access to trusted hardware. They bypass this impossibility result by assuming that all parties have access to both trusted hardware as well some additional set-up. The resulting protocols are more intricate and less efficient than ours, so our results can be interpreted as a practice-oriented approach to the security of the most natural MPC protocol that relies on SGX, which trades composability for efficiency while still preserving strong privacy guarantees for the inputs to the computation.

The overarching goal of our work is shared with the rich literature on software-only multiparty secure computation. In Appendix A we refer the works that are close to ours in the sense that they aim to bring secure multiparty computation to practice.

In Appendix C we recap the formal definitions for standard the cryptographic primitives we use in this work, including the partially authenticated notion of key exchange specialized for attestation settings introduced in [3].

2 IEEs, Programs, and Machines

The models that we develop in this paper rely on the abstraction for IEEs introduced in [3]. Here we recall the key features of that model. A more in depth description of these formalisms is provided in Appendix B.

An IEE is viewed as an idealised machine running some fixed program P and which exposes an interface through which one can pass inputs and receive outputs to/from P. The I/O behaviour of a process running in an IEE is determined by the program it is running, and the inputs it receives. The interface models the strict isolation between processes running in different IEEs and formalizes that the only information that is revealed about a program running within an IEE is contained in its input-output behaviour.

PROGRAMS. We extend the model for programs from [3] to the setting where inputs/outputs are labeled: programs are transition functions which take a current state st and a label-input pair (l, i), and produce a new output o and an updated state. We write $o \leftarrow P[\mathsf{st}](l, i)$ for each such action and refer to it as an *activation*. Throughout the paper we restrict our attention to programs (even if they are adversarially created) for which the transition function is guaranteed to run in polynomial-time. Programs are assumed to be deterministic modulo of system calls; in particular we assume a system can call rand for providing programs with fresh randomness. Additionally, outputs are assumed to include a flag $\mathsf{finished}$ that indicating if the transition function will accept further input. We extend our notation to account for probabilistic programs that invoke the rand system call. We write $o \leftarrow P[\mathsf{st}; r](l, i)$ for the activation of P which when invoked on labeled input (l, i) (with internal state st and random coins r) produced output o. We write a sequence of activations as $(o_1, \ldots, o_n) \leftarrow P[\mathsf{st}; r](l_1, i_1, \ldots, l_n, i_n)$ and denote by $\mathsf{Trace}_{P[\mathsf{st};r]}(l_1, i_1, \ldots, l_n, i_n)$ the corresponding input/output trace $(l_1, i_1, o_1, \ldots, l_n, i_n, o_n)$. For a trace T, we write $\mathsf{filter}[L](T)$ for the projection of the trace that retains only I/O pairs that correspond to labels in L. We use $\mathsf{filter}[l]$ when L is a singleton. We also extend the basic notion of program composition in [3] to consider label-based parallel and sequential program composition. Intuitively, when two labelled programs are composed, the set of labels of the composed program is enriched to encode the precise sub-program that should be activated and the label on which it should be activated.

MACHINES. As in [3] we model machines via a simple external interface, which we see as both the functionality that higher-level cryptographic schemes can rely on when using the machine, and the adversarial interface that will be the basis of our attack models. This interface can be thought of as an abstraction of Intel's SGX [30]. The interface consists of three calls: 1. $\mathsf{Init}(1^\lambda)$ initialises the machine and outputs the global parameters prms. 2. $\mathsf{Load}(P)$ loads the program P in a fresh IEE and returns its handle hdl 3. $\mathsf{Run}(\mathsf{hdl}, l, i)$ passes the label-input pair (l, i) to the IEE with handle hdl. We define the I/O trace $\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl})$ of a process hdl running in a machine $\mathcal{M}$ as the tuple $(l_1, i_1, o_1, \ldots, l_n, i_n, o_n)$ that includes the entire sequence of n inputs/outputs resulting from all invocations of Run on hdl; $\mathsf{Program}_{\mathcal{M}}(\mathsf{hdl})$ is the code (program) running under handle hdl; $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl})$ represents the coins given to the program by the rand system call; and $\mathsf{State}_{\mathcal{M}}(\mathsf{hdl})$ is the internal state of the program. Finally, we will write $\mathcal{A}^{\mathcal{M}}$ to indicate that algorithm $\mathcal{A}$ has access to machine $\mathcal{M}$.

3 Labelled Attested Computation

We now formalize a cryptographic primitive that generalizes the notion of Attested Computation proposed in [3], called Labelled Attested Computation. The main difference to the original proposal is that, rather than fixing a particular form of program composition for attestation, Labelled Attested Computation is agnostic of the program's internal structure; on the other hand, it permits controlling data flows and attestation guarantees via the label information included in program inputs.

SYNTAX. A *Labelled Attested Computation* (LAC) scheme is defined by the following algorithms:

- $\mathsf{Compile}(\mathsf{prms}, P, L^*)$ is the deterministic program compilation algorithm. On input global parameters for some machine $\mathcal{M}$, program P and an attested label set L^*, it outputs program P^*. This algorithm is run locally. P^* is the code to be run as an isolated process in the remote machine, whereas L^* defines which labelled inputs should be subject to attestation guarantees.
- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, l, i)$ is the stateless attestation algorithm. On input global parameters for $\mathcal{M}$, a process handle hdl and label-input pair (l, i), it uses the interface of $\mathcal{M}$ to obtain attested output o^*. This algorithm is run remotely, but in an unprotected environment: it is responsible for interacting with the isolated process running P^*, providing it with inputs and recovering the attested outputs that should be returned to the local machine.
- $\mathsf{Verify}(\mathsf{prms}, l, i, o^*, \mathsf{st})$ is the public (stateful) output verification algorithm. On input global parameters for $\mathcal{M}$, a label l, an input i, an attested output o^* and some state st it produces an output value o and an updated state, or the failure symbol $\perp$. This failure symbol is encoded so as to be distinguishable from a valid output of a program, resulting from a successful verification. This algorithm is run locally on claimed outputs from the Attest algorithm. The initial value of the verification state is set to be (prms, P, L^*), the same inputs provided to $\mathsf{Compile}$.

Intuitively, a LAC scheme is correct if, for any given program P and attested label set L^*, assuming an honest execution of all components in the scheme, both locally and remotely, the local user is able to accurately reconstruct a partial view of the I/O sequence that took place in the remote environment, for an arbitrary set of labels L. A formal definition of correctness is provided in Appendix D.

SECURITY. Security of labelled attested computation imposes that an adversary with control of the remote machine cannot convince the local user that some arbitrary remote (partial) execution of a program P has occurred, when it has not. It says nothing about the parts of the execution trace that are hidden from the client *or* are not in the attested label set L^*. Formally, we allow the adversary to freely interact with the remote machine, whilst providing a sequence of (potentially forged) attested outputs for a specific label $l \in L^*$. The adversary wins if the local user reconstructs an execution trace without aborting (i.e., all attested outputs must be accepted by the verification algorithm) and one of two conditions occur: i. there does not exist a remote process hdl^* running a compiled version of P where a consistent set of inputs was provided *for label* l; or ii. the outputs recovered by the local user for those inputs are not consistent with the semantics of P.

```
Game $\mathsf{Att}_{\mathsf{LAC},\mathcal{A}}(1^\lambda)$:
$\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$; $(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$; $P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$; $\mathsf{st}_V \leftarrow (\mathsf{prms}, P, L^*)$
For $k \in [1..n]$ :
  $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_2^\mathcal{M}(\mathsf{st}_\mathcal{A})$; $(o_k, \mathsf{st}_V) \leftarrow \mathsf{Verify}(\mathsf{prms}, l, i_k, o_k^*, \mathsf{st}_V)$
  If $o_k = \perp$ Return $\mathsf{F}$
$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For $\mathsf{hdl}^*$ s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
  $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$; $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
  If $T \sqsubseteq T'$ Return $\mathsf{F}$
Return $\mathsf{T}$
```

Fig. 1. Game defining the security of LAC.

Technically, these conditions are checked in the definition by retrieving the full sequence of *label-input pairs* and random coins passed to all compiled copies of P running in the remote machine and running P on the same inputs to obtain the expected outputs. One then checks that for at least one of these executions, when the traces are restricted to special label l, that the expected trace matches the locally recovered trace via Verify. Since the adversary is free to interact with the remote machine as it pleases, we cannot hope to prevent it from providing arbitrary inputs to the remote program at arbitrary points in time, while refusing to deliver the resulting (possibly attested) outputs to the local user. This justifies the winning condition referring to a prefix of the execution in the remote machine, rather than imposing trace equality. Indeed, the definition's essence is to impose that, if the adversary delivers attested outputs for a particular label in the attested label set, then the subtrace of verified outputs for that label will be an exact prefix of the projection of the remote trace for that label.

We note that a higher-level protocol relying on LAC can fully control the semantics of labels, as these depend on the semantics of the compiled program. In particular, adopting the specific forms of parallel and sequential composition presented in Section 2, it is possible to use labels to get the attested execution of a sub-program that is fully isolated from other programs that it is composed with. This provides a much higher degree of flexibility than that offered by the original notion of Attested Computation.

Definition 1 (Security). *A labelled attested computation scheme is secure if, for all ppt adversaries $\mathcal{A}$, the probability that experiment in Fig. 1 returns* T *is negligible.*

The adversary loses the game if there exists at least one remote process that matches the locally reconstructed trace. This should be interpreted as the guarantee that IEE resources are indeed being allocated in a specific remote machine to run at least one instance of the remote program (note that if the program is deterministic, many instances could exist with exactly the same I/O behaviour, which is *not* seen as a legitimate attack). Furthermore, our definition imposes that the compiled program uses essentially the same randomness as the source program (except of course for randomness that the security module internally uses to provide its cryptographic functionality), as otherwise it is easy for the adversary to make the (idealized) local trace diverge from the remote. This is a consequence of our modelling approach, but it does not limit the applicability of our primitive: it simply spells out that the transformation performed on the code for attestation will typically consist of an instrumentation of the code by applying cryptographic processing to the inputs and outputs it receives.

MINIMAL LEAKAGE. The above discussion shows that a LAC scheme guarantees that the I/O behaviour of the program in the remote machine includes at least the information required to reconstruct an hypothetical local execution of the source program. Next, we require that a compiled program does not reveal any information beyond what the original program would reveal. We achieve this by imposing that our LAC schemes satisfy the minimal leakage definition from [3] (recalled in Appendix D) which ensures that nothing from the internal state of the source programs (in addition to what is public, i.e., the code and I/O sequence) is leaked in the trace of the compiled program.

4 LAC from SGX-like systems

Our labelled attested computation protocol relies on the capabilities offered by the security module of Secure Guard Extensions (SGX) architecture proposed by Intel [2] (i.e. MACs for authenticated communication between IEEs, and digital signatures for inter-platform attestation of executions). Our security module formalization is the same as the one adopted in [3].

SECURITY MODULE. The security module relies on a signature scheme and a MAC scheme,and operates as follows:

- On initialization, the security module generates a key pair $(\mathsf{pk}, \mathsf{sk})$ and a symmetric key key.It also creates a special process running code S^* in an IEE with handle 0. The security module then securely stores the key material, and outputs the public key.
- The operation of IEE with handle 0 is different from all other IEEs in the machine. Program S^* will permanently reside in this IEE, and it will be the only one with direct access to both sk and key. The code of S^* is dedicated to transforming messages authenticated with key into messages signed with sk. On activation, it expects an input (m, t). It obtains key from the security module and verifies the tag. If the previous operation was successful, it obtains sk from the security module, signs the message and outputs the signature.
- The security module exposes a single system call $\mathsf{mac}(\mathsf{m})$ to code running in all other IEEs. On such a request from a process running program P, the security module returns a MAC tag t computed using key over both the code of P and the input message m.

LABELLED ATTESTED COMPUTATION SCHEME. We now define a LAC scheme that relies on a remote machine supporting such a security module. Basic replay protection using a sequence number does not suffice to bind a remote process to a subtrace, since the adversary could then run multiple copies of the same process and *mix and match* outputs from various traces. This is similar to the reasoning in [3]. However, in this paper we are interested in validating traces for specific attested labels, independently from each other, rather than the full remote trace. Our LAC scheme works as follows:

- $\mathsf{Compile}(\mathsf{prms}, P, L)$ generates a new program P^* and outputs it. Program P^* is instrumented as follows:
  - in addition to the internal state st of P, it maintains a list ios_l of all the I/O pairs it has previously received and computed for each label $l \in L$.
  - On input (l, i), P^* computes $o \leftarrow\$ P[\mathsf{st}_P](l, i)$ and verifies if $l \in L$. If this is not the case, then P^* simply outputs non-attested output o.

  - Otherwise, it updates the list ios by appending (l, i, o), computes the subset of ios for label l : $\mathsf{ios}_l \leftarrow \mathsf{filter}[l](\mathsf{ios})$ and requests from the security module a MAC of for that list. Due to the operation of the security module, this will correspond to a tag t on the tuple (P^*, ios_l).
  - It finally outputs $(o, \mathsf{t}, P^*, \mathsf{ios}_l)$. We note that we include (P^*, ios_l) explicitly in the outputs of P^* for clarity of presentation only. This value would be kept in an insecure environment by a stateful Attest program.

– $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, l, i)$ invokes $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, (l, i))$ using the handle and input value it has received. Attest then checks is the produced output o is to be attested and if so transforms the tag into a signature σ using the IEE with handle 0 and outputs (o', σ). Otherwise it simply outputs o.

– $\mathsf{Verify}(\mathsf{prms}, l, i, o^*, \mathsf{st})$ is the stateful verification algorithm. Initially $\mathsf{st} = (\mathsf{prms}, P, L^*)$, on first activation Verify computes and stores P^* and initialises an empty list ios of input-output pairs. Verify returns o^* if $l \notin L$. Otherwise, it first parses o^* into (o, σ), appends (l, i, o) to ios and verifies the digital signature σ using prms and $(P^*, \mathsf{filter}[l](\mathsf{ios}))$. If parsing or verification fails, Verify outputs $\perp$. If not, then Verify outputs o.

It is easy to see that our LAC scheme is correct. A precise argument for correctness is provided in Appendix D.

Theorem 1 (LAC scheme security). *The LAC scheme presented above provides secure attestation if the underlying MAC scheme Π and signature scheme Σ are existentially unforgeable. Furthermore, it unconditionally ensures minimum leakage.*

The full proof of this theorem can be found in Appendix E. The proof intuition is a generalization of the argument for the attested computation scheme in [3]. All attested outputs are bound to a partial execution trace that contains the entire I/O sequence associated with the corresponding attested label, so all messages accepted by Verify must exist as a prefix for a remote trace of some instance of P^*. The adversary can only cause an inconsistency in $T \sqsubseteq T'$ if the signature verification performed by Verify accepts a message of label $l \in L^*$ that was never authenticated by an IEE running P^*. However, in this case the adversary is either breaking the MAC scheme (and dishonestly executing Attest), or breaking the signature (directly forging attested outputs).

5 Secure computation with IEEs

FUNCTIONALITIES. We want to securely execute a functionality $\mathcal{F}$ defined by a four-tuple $(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout})$, where F is a deterministic stateful transition function that takes inputs of the form (id, i). Here, id is a party identifier, which we assume to be an integer in the range $[1..n]$, and n is the total number of participating parties. On each transition, F produces an output that is intended for party id, as well as an updated state. We associate to F two leakage functions $\mathsf{Lin}(k, i, \mathsf{st})$ and $\mathsf{Lout}(k, o, \mathsf{st})$ which define the public leakage that can be revealed by a protocol about a given input i or output o for party k, respectively; for the sake of generality, both functions may depend on the internal state st of the functionality, although this is not the case in the examples we consider in this paper. Arbitrary reactive functionalities formalized in the Universal Composability framework can be easily recast as a transition function such as this. The upside of our approach is that one obtains a precise code-based definition of what the functionality should do (this is central to our work since these descriptions give rise to concrete programs); the downside is that the code-based definitions may be less clear to a human reader, as one cannot ignore the tedious *book-keeping* parts of the functionality.

EXECUTION MODEL. We assume the existence of a machine $\mathcal{M}$ allowing for the usage of isolated execution environments, as defined in Section 2. In secure computation terms, this machine should *not* be seen as an ideal functionality that enables some hybrid model of computation, but rather an additional party that comes with a specific setup assumption, a fixed internal operation, and which cannot be corrupted. Importantly, all interactions with $\mathcal{M}$ and all the code that is run in $\mathcal{M}$ but outside IEEs is considered to be adversarially controlled.

SYNTAX. A protocol π for functionality $\mathcal{F}$ is a seven-tuple of algorithms as follows:

- Setup – This is the party local set-up algorithm. Given the security parameter, the public parameters prms for machine $\mathcal{M}$ and the party's identifier id, it returns the party's initial state st (incluing its secret key material) and its public information pub.
- $\mathsf{Compile}$ – This is the (deterministic) code generation algorithm. Given the description of a functionality F, and public parameters $(\mathsf{prms}, \mathsf{Pub})$ for both the remote machine and the entire set of public parameters for the participating parties, it generates the instrumented program that will run inside an IEE.
- Remote – This is the untrusted code that will be run in $\mathcal{M}$ and which ensures the correctness of the protocol by controlling its scheduling and input collection order. It has oracle access to $\mathcal{M}$ and is in charge of collecting inputs and delivering outputs. Its initial state describes the order in which inputs of different parties should be provided to the functionality.
- Init – This is the party local protocol initialization algorithm. Given the party's state st produced by Setup and the public information of all participants Pub it outputs an uptated state st. We note that a party can choose to engage in a protocol by checking if the public parameters of all parties are correct and assigned to roles in the protocol that match the corresponding identities.
- $\mathsf{AddInput}$ – This is the party local input providing algorithm. Given the party's current state st and an input in, it outputs an uptated state st.
- $\mathsf{Process}$ – This is the party local message processing algorithm. Given its internal state st, and an input message m, it runs the next protocol stage, updates the internal state and returns output message m'.
- Output – This is the party local output retrieval algorithm. Given internal state st, it returns the current output o.

Intuitively such a protocol is correct if it can support any execution schedule whilst evaluating the functionality correctly. A precise definition is provided in Appendix F.

SECURITY. As is customary in secure computation models, we take the ideal world versus real world approach to define security of a protocol. Our security model is presented in Figure 2, and is described as follows. In the real world, the adversary interacts with

```
Game Real_{F,π,A,M}(1^λ):
  $(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
  $\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
  $(\mathsf{st}_\mathcal{A}, k) \leftarrow\$ \mathcal{A}(\mathsf{prms})$
  For id $\in [1..k]$:
    $(\mathsf{st}_\mathsf{id}, \mathsf{pub}_\mathsf{id}) \leftarrow\$ \mathsf{Setup}(\mathsf{prms}, \mathsf{id})$
  $\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_k)$
  For id $\in [k+1..n]$:
    $(\mathsf{st}_\mathcal{A}, \mathsf{pub}_\mathsf{id}) \leftarrow\$ \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{id}, \mathsf{Pub})$
  $\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_n)$
  For id $\in [1..k]$:
    $\mathsf{st}_\mathsf{id} \leftarrow\$ \mathsf{Init}(\mathsf{st}_\mathsf{id}, \mathsf{Pub})$
  $b \leftarrow\$ \mathcal{A}^\mathcal{O}(\mathsf{st}_\mathcal{A})$

Oracle Send(id, $m$):
  If id $\notin [1..k]$ Return $\perp$
  $(\mathsf{st}_\mathsf{id}, m') \leftarrow\$ \mathsf{Process}(\mathsf{st}_\mathsf{id}, m)$
  Return $m'$

Oracle SetInput(in, id):
  If id $\notin [1..k]$ Return $\perp$
  $\mathsf{st}_\mathsf{id} \leftarrow\$ \mathsf{AddInput}(\mathsf{in}, \mathsf{st}_\mathsf{id})$

Oracle Load($P$):
  Return $\mathcal{M}.\mathsf{Load}(P)$

Oracle Run(hdl, $l$, $x$):
  Return $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, l, x)$

Oracle GetOutput(id):
  If id $\notin [1..k]$ Return $\perp$
  Return $\mathsf{Output}(\mathsf{st}_\mathsf{id})$
```

```
Game Ideal_{F,π,A,S}(1^λ):
  $(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
  $\mathsf{st}_\mathsf{F} \leftarrow \epsilon$
  $(\mathsf{st}, \mathsf{prms}) \leftarrow\$ \mathcal{S}(1^\lambda)$
  $(\mathsf{st}_\mathcal{A}, k) \leftarrow\$ \mathcal{A}(\mathsf{prms})$
  For id $\in [1..k]$:
    $(\mathsf{st}, \mathsf{pub}_\mathsf{id}) \leftarrow\$ \mathcal{S}(\mathsf{st}, \mathsf{id})$
    $\mathsf{ListIn}_\mathsf{id} \leftarrow [\,]$
    $\mathsf{ListOut}_\mathsf{id} \leftarrow [\,]$
  $\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_k)$
  For id $\in [k+1..n]$:
    $(\mathsf{st}_\mathcal{A}, \mathsf{pub}_\mathsf{id}) \leftarrow\$ \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{id}, \mathsf{Pub})$
  $\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_n)$
  For id $\in [1..k]$:
    $\mathsf{st} \leftarrow\$ \mathcal{S}(\mathsf{st}, \mathsf{id}, \mathsf{Pub})$
  $b \leftarrow\$ \mathcal{A}^\mathcal{O}(\mathsf{st}_\mathcal{A})$

Oracle Fun(id, in):
  If id $\in [1..k]$:
    $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{ListIn}_\mathsf{id}$
    $\mathsf{ListIn}_\mathsf{id} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_{k\text{-}1})$
    $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_\mathsf{F}](\mathsf{id}, \mathsf{in}_k)$
    $\mathsf{ListOut}_\mathsf{id} \leftarrow \mathsf{out} : \mathsf{ListIn}_\mathsf{id}$
    Return $\mathsf{Lout}(\mathsf{out}, \mathsf{id}, \mathsf{st}_\mathsf{F})$
  Else
    $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_\mathsf{F}](\mathsf{id}, \mathsf{in})$
    Return out

Oracle SetInput(in, id):
  If $i \notin [1..k]$ Return $\perp$
  $\ell \leftarrow \mathsf{Lin}(\mathsf{in}, \mathsf{id}, \mathsf{st}_\mathsf{F})$
  $\mathsf{st} \leftarrow\$ \mathcal{S}(\mathsf{st}, \ell, \mathsf{id})$
  $\mathsf{ListIn}_\mathsf{id} \leftarrow \mathsf{in} : \mathsf{ListIn}_\mathsf{id}$

Oracle Send(id, m):
  $(\mathsf{st}, \mathsf{out}) \leftarrow\$ \mathcal{S}^\mathsf{Fun}(\mathsf{st}, \mathsf{id}, \mathsf{m})$
  Return out

Oracle Load($P$):
  $(\mathsf{st}, \mathsf{out}) \leftarrow\$ \mathcal{S}(\mathsf{st}, P)$
  Return out

Oracle Run(hdl, $l$, $x$):
  $(\mathsf{st}, \mathsf{out}) \leftarrow\$ \mathcal{S}^\mathsf{Fun}(\mathsf{st}, \mathsf{hdl}, l, x)$
  Return out

Oracle GetOutput(id):
  If id $\notin [1..k]$ Return $\perp$
  $i \leftarrow\$ \mathcal{S}(\mathsf{st}, \mathsf{id})$
  $(\mathsf{out}_1, \ldots, \mathsf{out}_k) \leftarrow \mathsf{ListOut}_\mathsf{id}$
  Return $\mathsf{out}_1 \,||\, \ldots \,||\, \mathsf{out}_i$
```

Fig. 2. Real and Ideal security games.

an IEE-enabled machine $\mathcal{M}$ under adversarial control and oracles SetInput, GetOutput and Send providing it with the locally run part of the protocol. In the ideal world, the adversary is presented with 1. a simulator $\mathcal{S}$ emulating the remote machine, the setup phase, and the Send oracle 2. idealised oracles SetInput, GetOutput. The idealised oracles only do book-keeping of which input should be transmitted next and which output should be retrieved next for each honest party. $\mathcal{S}$ gets given oracle access to a functionality evaluation oracle Fun that consumes the next input of a party (defined in SetInput if the party is honest, passed as input otherwise) and sets the next output for this party, returning the leakage of the input and output if the party is honest, and the full output otherwise. A protocol is deemed secure if there exists a ppt simulator such that no ppt adersary can distinguish the two worlds.

Definition 2. *We say π is secure for $\mathcal{F}$ if, for any ppt adversary $\mathcal{A}$, there exists a ppt simulator $\mathcal{S}$ such that the following definition of advantage is a negligible function in the security parameter.*

$$|\,\Pr[\mathsf{Real}^{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda) \Rightarrow b = 1] - \Pr[\mathsf{Ideal}^{\mathcal{F},\pi,\mathcal{A},\mathcal{S}}(1^\lambda) \Rightarrow b = 1]\,|$$

Succinctly, our model is inspired in the UC framework, and can be derived from it when natural restrictions are imposed: PKI, static corruptions, and a distinguished noncorruptible party modeling an SGX-enabled machine.[5] A security proof for a protocol in our model can be interpreted as translation of any attack against the protocol in the real world, as an attack against the ideal functionality in the ideal world. The simulator performs this translation by presenting an execution environment to the adversary that is consistent with what it is expecting in the real world. It does this by simulating the operations of the Load, Run and Send oracles, which represent the operation of honest parties in the protocol. While the adversary is able to provide the inputs and read the outputs for honest parties directly from the functionality, the simulator is only able obtain partial leakage about this values via the Lin and Lout functions. Conversely, it can obtain the functionality outputs for corrupt parties via the Fun oracle and, furthermore, it is also able to control the rate and order in which all inputs are provided to the functionality. Were this not the case, the adversary would be able to distinguish the two worlds by manipulating scheduling in a way the simulator could not possibly match.

6 A New MPC Protocol from SGX

In this section we describe our secure multiparty computation protocol based on LAC that works for any functionality. The protocol starts by running bootstrap code in an isolated execution environment in the remote machine; the code exchanges keys with each of the participants in the protocol. These key exchange programs are composed in parallel, as seen in Section 2. We reuse the notion of AttKE (key exchange for attested computation) from [3] which provides the right notion of key exchange security in this context. We recall the definition of AttKE security in Appendix C. Once this bootstrap stage is concluded, the code of the functionality starts executing. The functionality uses the secure channels established before to ensure that the inputs and outputs are private and authenticated. The security of this protocol relies on a utility theorem similar to that of [3] for the use of key exchange in the context of attestation. This theorem shows that, under the specific program composition pattern that we require for our MPC protocol, which guarantees AttKE isolation from other programs, each party obtains a secret key that is indistinguishable from a random string. It follows that the key can be used to construct a secure channel that connects it to code emulating the functionality within an IEE. Details are provided in Appendix G.

BOXING USING AUTHENTICATED ENCRYPTION. As explained above, after the bootstrapping stage of our protocol, we run the ideal functionality within an isolated execution environment. We implement this part of the execution using the *boxing* construction shown in Figure 3. The name comes by analogy with placing the functionality within a box, which parties can access via secure channels. The labelled program $\mathsf{Box}\langle\mathcal{F}, \Lambda\rangle$ is parametrized by a functionality $\mathcal{F}$ for n parties and a secure authenticated encryption encryption scheme Λ. Its initial state is assumed to contain n symmetric keys compatible with Λ, denoted sk_1 to sk_n (one for each participating party) and the empty initial state for the functionality st_F. The Box expects encrypted inputs i^* under a label l identifying the party providing the input. These are then decrypted using the respective key

[5] This particular choice in our model has implications for the composability properties of our results, as discussed in the related work section.

```
**Program** $\mathsf{Box}\langle\mathcal{F}, \Lambda\rangle[\mathsf{st}](i^*, l)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$\mathsf{id} \leftarrow l$
If $\mathsf{id} \notin [1..n]$ : Return $\perp$
If $\mathsf{st.seq}_{\mathsf{id}} = \epsilon$ : $\mathsf{st.seq}_{\mathsf{id}} \leftarrow 0$
$i \leftarrow \Lambda.\mathsf{Dec}(\mathsf{st.key}_{\mathsf{id}}, \mathsf{m})$
If $\mathsf{m} \neq (\mathsf{in}, \mathsf{st.seq}_{\mathsf{id}})$ : Return $\perp$
$o \leftarrow F[\mathsf{st.st}_{\mathsf{F}}](\mathsf{id}, \mathsf{in})$
$\mathsf{st.seq}_{\mathsf{id}} \leftarrow \mathsf{st.seq}_{\mathsf{id}} + 1$
$c \leftarrow\!\$\ \Lambda.\mathsf{Enc}(\mathsf{st.key}_{\mathsf{id}}, (\mathsf{seq}, o))$
$\mathsf{st.seq}_{\mathsf{id}} \leftarrow \mathsf{st.seq}_{\mathsf{id}} + 1$
Return $c$
```

Fig. 3. Boxing using Authenticated Encryption

sk_l and provided to $\mathcal{F}$. The value returned by the functionality is encrypted using the same sk_l and is then returned. To avoid replays of encrypted messages, we keep one sequence number $\mathsf{seq}_{\mathsf{id}}$ per communicating party id.

THE PROTOCOL. Building on top of a LAC scheme, an AttKE scheme and our Box construction we define a general secure multiparty computation protocol that works for any (possibly reactive) functionality F. The core of the protocol is the execution of an AttKE for each participant in parallel, followed by the execution of the functionality F on the remote machine, under a secure channel with each participant as specified in the Box construct. More precisely:

- Setup derives the code for a remote key exchange program $\mathsf{Rem}_{\mathsf{KE}}$ using the AttKE setup procedure. This code (which intuitively includes cryptographic public key material) is set to be the public information for this party. The algorithm also stores various parameters in the local state for future usage.
- Compile uses the LAC compilation algorithm on a program that results from the parallel composition of all the remote key exchange programs for all parties, which is then sequentially composed with the boxed functionality.
- Init locally recomputes the program that is intended for remote execution, as this is needed for attestation verification. The set of labels that define the locally recovered trace is set to those of the form $((p, (\mathsf{id}, \epsilon)), (q, \mathsf{id}))$, which correspond to those exactly matching the parts of the remote trace that are relevant for this party, namely its own key exchange and its own inputs/outputs.
- Process is split into two stages. In the first stage it uses LAC with attested labels of the form $(p, (\mathsf{id}, \epsilon))$ to execute AttKE protocol and establish a secure channel with the remote program. In the second stage, it uses non-attested labels of the form (q, id), and it provides inputs to the remote functionality (on request) and recovers the corresponding outputs when they are delivered.
- Output reads the output in the state of the participant and returns it.
- AddInput adds an input to the end of the list of inputs that have to be transmitted by the participant.

Pseudo code of the protocol as well as details of the (untrusted) scheduling algorithm can be found in Appendix G.

For proving security, we restrict the functionalities we consider to a particular leakage function: size of inputs/outputs. We say that a functionality $(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout})$ leaks size if it is such that Lin and Lout return the length of the inputs/outputs ($\mathsf{Lin}(k, x, \mathsf{st}) = \mathsf{Lout}(k, x, \mathsf{st}) = |x|$ for every k, x, st).

Theorem 2. *If* LAC *is a correct and secure LAC scheme,* AttKE *is a secure AttKE scheme and Λ a secure authenticated encryption scheme, then the protocol described in this Section is correct and secure for any functionality that leaks size.*

PROOF SKETCH. We build the required simulator $\mathcal{S}$ as follows. For dishonest parties, the simulator executes the protocol normally while for the honest parties instead of encrypting the inputs/outputs the simulator encrypts dummy messages of the correct length (obtained through the leakage function) under freshly generated keys.

We sketch a proof of indistinguishability between the real world and this ideal world. A detailed proof can be found in Appendix I. The proof is performed in 3 hops, the first corresponds to a hybrid argument over the honest parties in the protocol. In this hybrid argument one gradually replaces the secret key derived by each honest party by a random one. In each step, the AttKE utility theorem can be used to show that this change cannot be noticed by the adversary. In the second hop, we replace the encrypted inputs/outputs for honest parties by encrypted dummy payloads of the correct length. This hop is correct by the indistinguishability of authenticated encryption ciphertexts. After this last game hop, the resulting game is *identical until bad* to the ideal world, where the bad event corresponds to the simulator aborting due to an inconsistent message being accepted as the next undelivered input or output. Due to the use of sequence numbers, this bad event can be reduced to the authenticity of the encryption scheme and the Theorem follows.

7 Implementation

We provide an implementation of our protocol sgx-mpc-nacl relying on NaCl for the cryptographic library and Intel SGX for the IEEs. We make use of elliptic curves for both the key exchange (Diffie-Hellman) and digital signatures, and a combination of the Salsa20 and Poly1305 encryption and authentication schemes [9] for authenticated encryption. Our implementation relies on Intel's Software Development Kit (SDK) for dealing with the SGX low-level operations. These include loading code into an IEE (our Load abstraction), calling a top-level function within the IEE (our Run abstraction), and constructing an attested message (first getting a MAC'ed message within the IEE, and using the quoting enclave to convert it into a digital signature). It employs the LAC scheme proposed in this paper, and include wrappers that match our abstractions of digital signatures and authenticated encryption. These are then used to construct the secure bootstrapping protocol (AttKE) that enables each party to establish an independent secret key, and a secure channel that uses this key to communicate with the Box construction running inside the enclave. Finally, our implementation of the Box is agnostic of the functionality that should be computed by the protocol, and can be linked to arbitrary functionality implementations, provided that these comply with a simple labelled I/O interface. The top-level interface to our protocol includes the code that should be run inside the IEE, the code that runs outside the IEE in the remote machine to perform the book-keeping operations and the client-side code that permits bootstrapping a secure channel and then send inputs/receive outputs from the functionality.

We compare our implementation with measurements we performed using the ABY framework [21]. We chose ABY for comparison, as we could evaluate it on the same platform we used for assessing our protocol, therefore avoiding differences due to performance disparities of heterogeneous evaluation platforms. Although it is specific to the two-party secure computation setting, ABY is representative of state-of-the-art MPC implementations and we expect results for other frameworks such as Sharemind [18] and SPDZ [19] to lead to similar conclusions; indeed, the crux of our performance gains resides in the fact that our solution does not require encoding the computation in circuit form, unlike all the aforementioned protocols.[6]

Like our protocol, the ABY protocol has two phases: a *preparation phase* and an *online phase*. The preparation phase comprises the key exchange between the input parties by means of oblivious transfer (OT), and generation of the garbled circuit (GC) representing the desired function. In the online phase the GC gets evaluated and the result are send back to the output party. In our protocol, the preparation phase is used to establish a secure channel between the IEE and the input parties. The online phase of our protocol comprises the decryption of inputs in the Box component, the evaluation of the payload function, and the encryption of the results, again by the Box component.

We evaluated the performance of four different secure two-party computation use cases (Table 1): AES, millionaire's problem, private set intersection and hamming distance. In comparison to ABY, the preparation phase and online phase are shorter with sgx-mpc-nacl, and consequently the overall runtime is faster as well. In general, sgx-mpc-nacl is faster for all the testing computations performed. However, the gains are considerably more noticeable when we increase with input size and computation. This has the highest significance on evaluation of the private set intersection with the largest input size (1 mill.), where our implementation is roughly 300 times faster.

Table 1. Clockwisely, starting from upper left: hamming distance, AES, millionaire's problem and private set intersection

	Phase	**Preparation** (ms)		**Online** (ms)		**Total** (ms)	
	Protocol	**ABY**	**Ours**	**ABY**	**Ours**	**ABY**	**Ours**
Input size ($bits$)	*160*	196.3	115.7	0.752	0.050	197.1	117.75
	1600	196.7	115.7	1.819	0.302	198.5	116.00
	16000	201.6	115.7	13.14	2.798	214.7	118.50
	160000	226.2	115.2	144.4	27.77	370.6	142.97

Phase	**ABY**	**Ours**
Preparation (ms)	197.9	115.84
Online (ms)	3.249	0.661
Total (ms)	201.1	116.50

	Phase	**Preparation**		**Online**		**Total**	
	Protocol	**ABY**	**Ours**	**ABY**	**Ours**	**ABY**	**Ours**
Set size	*100*	224.8	115.8	1.084	0.043	225.9	115.84
	1000	368.1	115.8	2.168	0.199	370.3	116.00
	10,000	1442.2	115.8	12.88	1.758	1455.1	117.56
	100,000	10,698.7	115.7	109.5	17.39	10,808.2	133.09
	1,000,000	84,096.6	115.7	1616.0	173.1	85,712.6	288.80

Phase	**ABY**	**Ours**
Preparation (ms)	196.3	127.7
Online (ms)	0.404	0.024
Total (ms)	196.7	127.7

In Appendix J we discuss the implications of side-channel vulnerabilities on implementations over IEE-enabled systems generally and Intel's SGX in particular, and in what way our implementation can protect against such attacks.

[6] We also note that ABY assumes a semi-honest adversary, which is weaker than the one we consider; but still our performance gains are significant.

References

1. J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, Aug. 2016. USENIX Association.
2. I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP 2013*, page 10, 2013.
3. M. Barbosa, B. Portela, G. Scerri, and B. Warinschi. Foundations of hardware-based attested computation and application to SGX. In *EuroS&P*, pages 245–260. IEEE, 2016.
4. M. Barbosa, B. Portela, G. Scerri, and B. Warinschi. Foundations of hardware-based attested computation and application to SGX. *IACR Cryptology ePrint Archive*, 2016:14, 2016.
5. A. Baumann, M. Peinado, and G. C. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, pages 267–283. USENIX Association, 2014.
6. M. Bellare, J. Kilian, and P. Rogaway. The security of cipher block chaining. In *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358. Springer, 1994.
7. A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008.
8. D. J. Bernstein. Cache-timing attacks on aes, 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
9. D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012.
10. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
11. P. Bogetoft, I. Damgård, T. P. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. In G. D. Crescenzo and A. D. Rubin, editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 142–147. Springer, 2006.
12. E. Brickell, L. Chen, and J. Li. A new direct anonymous attestation scheme from bilinear maps. In *TRUST*, volume 4968 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 2008.
13. E. F. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *ACM Conference on Computer and Communications Security*, pages 132–145. ACM, 2004.
14. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85. Springer, 2007.
15. R. Canetti and M. Fischlin. Universally composable commitments. In *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.
16. L. Catuogno, A. Dmitrienko, K. Eriksson, D. Kuhlmann, G. Ramunno, A. Sadeghi, S. Schulz, M. Schunter, M. Winandy, and J. Zhan. Trusted virtual domains - design, implementation and lessons learned. In *INTRUST*, volume 6163 of *Lecture Notes in Computer Science*, pages 156–179. Springer, 2009.
17. V. Costan and S. Devadas. Intel SGX explained, 2016.
18. CYBERNETICA. Sharemind. `https://sharemind.cyber.ee/`. [Online; accessed 14-November-2016].
19. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

20. I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft. Confidential benchmarking based on multiparty computation. *IACR Cryptology ePrint Archive*, 2015:1006, 2015.
21. D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*. The Internet Society, 2015.
22. A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. In *DATE*, pages 1–6. European Design and Automation Association, 2014.
23. H. Ge and S. R. Tate. A direct anonymous attestation scheme for embedded devices. In *Public Key Cryptography*, volume 4450 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2007.
24. C. Gebhardt and A. Tomlinson. Secure virtual disk images for grid computing. In *APTC '08*, pages 19–29. IEEE Computer Society, 2008.
25. S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.
26. D. Gupta, B. Mood, J. Feigenbaum, K. R. B. Butler, and P. Traynor. Using intel software guard extensions for efficient two-party secure function evaluation. In *FC Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, volume 9604 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2016.
27. S. Halevi, Y. Lindell, and B. Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 132–150. Springer, 2011.
28. W. Henecka, S. Kögl, A. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *ACM Conference on Computer and Communications Security*, pages 451–462. ACM, 2010.
29. M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ISCA*, page 11. ACM, 2013.
30. Intel. *Software Guard Extensions Programming Reference*, 2014. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.
31. J. Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 115–128. Springer, 2007.
32. J. Katz and M. Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In *FSE*, volume 1978 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2000.
33. P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *EuroSys*, pages 10:1–10:14. ACM, 2014.
34. A. Langley. Lucky thirteen attack on TLS CBC. Imperial Violet, Feb. 2013. `https://www.imperialviolet.org/2013/02/04/luckythirteen.html`, Accessed October 25th, 2015.
35. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.
36. J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *EuroSys*, pages 315–328. ACM, 2008.
37. Microsoft. *BitLocker Drive Encryption: Data Encryption Toolkit for Mobile PCs: Security Analysis*, 2007. `https://technet.microsoft.com/en-us/library/cc162804.aspx`.
38. J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*, pages 479–494. USENIX Association, 2013.

39. R. Pass, E. Shi, and F. Tramer. Formal abstractions for attested execution secure processors. Cryptology ePrint Archive, Report 2016/1027, 2016. `http://eprint.iacr.org/2016/1027`.
40. F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy*, pages 38–54. IEEE Computer Society, 2015.
41. B. Smyth, M. Ryan, and L. Chen. Direct anonymous attestation (DAA): ensuring privacy with corrupt administrators. In *ESAS*, volume 4572 of *Lecture Notes in Computer Science*, pages 218–231. Springer, 2007.
42. Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, pages 640–656. IEEE Computer Society, 2015.

A Further Related Work

Fairplay is a system originally developed to support two-party computation [35] and then extended to FairplayMP to support multiparty computation [7]: Fairplay implements a two party computation protocol in the manner suggested by Yao; FairplayMP is based on the Beaver-Micali-Rogaway protocol. Sharemind [10] is a secure service platform for data collection and analysis, employing a 3-party additive secret sharing scheme and provably secure protocols in the honest-but-curious security model with no more than one passively corrupted party. TASTY (Tool for Automating Secure Two-partY computations) is a tool suite addressing secure two-party computation in the semi-honest model [28] whose main feature allows to compile and evaluate functions not only using garbled circuits, but also homomorphic encryption schemes, at the same time. SPDZ [19] is a protocol for general multi party computations considering active adversaries and tolerating the corruption of $n-1$ out of the n parties, leveraging a pre-processing stage for exchanging randomness between participants, towards reducing communication requirements associated with the on-line stage.

The main advantage of our solution with respect to the previous systems is its efficiency both for the parties that provide inputs and collect outputs from the computation, and those that perform the computation. In all of the above solutions, the computation is distributed, and the communication load for parties performing the computation grows (with varying degrees of scalability) with the complexity of the computed functionality (often expressed as a circuit). In our solution, a single party (the owner of the IEE-enabled machine) performs the computation which is run essentially as fast as the program that computes it in the clear, so the overhead is reduced to establishing secure channels with all other participants (which is observable on the measurements presented in Section 7). For these parties, the overhead is a single key exchange, and then all the inputs and outputs are transferred using standard authenticated encryption. On the downside, although our protocol is secure in the presence of active adversaries, we only consider static corruptions and rely on a strong trust assumption in idealizing the IEE-enabled machine.

B IEE, Programs and Machines

We recall here the key features of programs, machines and IEEs from [3] and highlight the minor modifications we make to this model.

PROGRAMS. The programs are assumed to be written in some programming language $\mathcal{L}$ enriched with IEE system calls. These calls give access to different cryptographic functionalities offered by *security module* interface. The cryptography offered may differ between IEEs. The language $\mathcal{L}$ is assumed to be deterministic modulo the operation of system calls; in particular we assume a system call rand which gives access to fresh random coins sampled uniformly at random. It is important for our results that system calls cannot be used by a program to store additional implicit state that would escape our control. To this end, we impose that the results of system calls within an IEE can depend only on: i. an initially shared state that is defined when a program is loaded (e.g., the cryptographic parameters of the machine, and the code of the program); ii. the input explicitly passed on that particular call; and iii. fresh random coins. As a consequence of this, we may assume that system calls placed by different parts of a program are identically distributed, assuming that the same input is provided. This is particularly important when we consider program composition below.

We use the same model for programs as [3] (extended to the settings where inputs/outputs are labeled): transition functions which take a current state st and a label-input pair (l, i), and produce a new output o and an updated state. We write $o \leftarrow P[\mathsf{st}](l, i)$ for each such action and refer to it as an *activation*. Throughout the paper we restrict our attention to programs (even if they are adversarially created) for which the transition funtion is guaranteed to run in polynomial-time.Unless otherwise stated, st is assumed to be initially empty. We impose that every output produced by a program includes a Boolean flag $\mathsf{finished}$ that indicates whether the transition function will accept further input. We will denote by $o.\mathsf{finished}$ the value of this flag in some output o. The transition function may return arbitrary outputs until it produces an output where $\mathsf{finished} = \mathsf{T}$, at which point it can return no further output or change its state. Some programs may not use labels internally and, in that case, we simply pass it the empty string at the label input.

MACHINES. In [3] machines can be accessed through a simple interface abstracting the capabilities of real IEE enabled machines. We recall this interface:

- $\mathsf{Init}(1^\lambda)$ is the global initialisation procedure which, on input the security parameter, outputs the global parameters prms. This algorithm represents the machine's hardware initialisation procedure, which is out of the user's and the adversary's control. Intuitively, it initialises the internal security module, the internal state of the remote machine and returns any public cryptographic parameters that the security module releases. The global parameters of machines are assumed to be authenticated using external mechanisms, such as a PKI.
- $\mathsf{Load}(P)$ is the IEE initialisation procedure. On input a program/transition function P, the machine produces a fresh handle hdl, creates a new IEE with handle hdl, loads P into the new IEE and returns hdl. The machine interface does not provide direct access to either the internal state of an IEE nor to its randomness input. This means

that the only information that is leaked about internal state and randomness input is that revealed (indirectly) via the outputs of the program.

– $\mathsf{Run}(\mathsf{hdl}, l, i)$ is the process activation procedure. On input a handle hdl and a label-input pair (l, i), it will activate process running in isolated execution environment of handle hdl with (l, i) as the next input. When the program/transition function produces the next output o, this is returned to the caller.

B.1 Program composition

We extend the basic notion of program composition in [3] to consider the two general label-based forms of program composition shown in Fig. 4 that can be applied recursively and interchangeably to create arbitrarily complex programs in a modular way.

```
Program $\langle P_1 \mid \ldots \mid P_n \rangle_{p_1,\ldots,p_n}[\mathsf{st}](l, i)$:
If $\mathsf{st} = \epsilon$ :
  For $i \in [1..n]$ : $\mathsf{st.finished}.p_i \leftarrow \mathsf{F}$; $\mathsf{st}.p_i \leftarrow \epsilon$
If $(\wedge_{i=1}^{n} \mathsf{st.finished}.p_i)$ : Return $\epsilon$
If $\exists k \in [1..n]$ s.t. $l = (p_k, l')$ :
  If $\neg\mathsf{st.finished}.p_k$ :
    $o \leftarrow\$ P_k[\mathsf{st}.p_k](l', i)$
    $\mathsf{st.finished}.p_k \leftarrow o.\mathsf{finished}$
  Else: $o \leftarrow \epsilon$
Else: $o \leftarrow \perp$
Return $(\wedge_{i=1}^{n} \mathsf{st.finished}.p_i, o)$
```

```
Program $\langle P \,;\, Q \rangle_{\phi,p,q}[\mathsf{st}](l, i)$:
If $\mathsf{st} = \epsilon$ :
  $\mathsf{st.stage} \leftarrow 0$; $\mathsf{st.finished} \leftarrow \mathsf{F}$ $\mathsf{st.st}' \leftarrow \epsilon$
If $\mathsf{st.finished}$ : Return $\epsilon$
If $\mathsf{st.stage} = 0 \wedge l = (p, l')$ :
  $o \leftarrow\$ P[\mathsf{st.st}'](l', i)$
  If $o.\mathsf{finished}$ : $\mathsf{st.stage} \leftarrow 1$; $\mathsf{st.st}' \leftarrow \phi(\mathsf{st.st}')$
Else:
  If $\mathsf{st.stage} = 1 \wedge l = (q, l')$ :
    $o \leftarrow\$ Q[\mathsf{st.st}'](l', i)$
    $\mathsf{st.finished} \leftarrow o.\mathsf{finished}$
  Else: $o \leftarrow \perp$
Return $(\mathsf{st.stage}, \mathsf{st.finished}, o)$
```

Fig. 4. Parallel (left) and sequential (right) program composition.

By parallel composition of programs $P_1, \ldots P_n$, denoted $\langle P_1 \mid \ldots \mid P_n \rangle_{p_1,\ldots,p_n}$, we mean the transition function that takes inputs with extended labels of the form (p_i, l)[7]—here p_i are bitstrings used to identify the target program, where we assume $p_i \neq p_j$ for i, j distinct—and dispatches incoming label-input pairs to the appropriate program. In parallel composition we exclude the possibility of state sharing between programs, and define termination to occur when all composed programs have terminated. By sequential composition of two programs P and Q via projection function ϕ, denoted $\langle P; Q \rangle_{\phi,p,q}$, we mean the transition function that has two execution stages, which are signaled in its output via an additional stage flag. As above, we will denote by $o.\mathsf{stage}$ the value of this flag in some output o. For consistency, we again assume labels of the form (p, l) and (q, l) where $p \neq q$ are used to identify the target program. In the first stage, every label-input pair will be checked for consistency (i.e., that it indicates P as the target program) and dispatched to program P. This will proceed until P's last output indicates it has finished (inclusively). The next activation will trigger the start of the second stage, at which point the composed program initialises the state of Q using $\phi(\mathsf{st}_P)$ before activating it for the first time. Additionally we require that a constant indicating the current stage is appended to any output of a composition. We do not admit other state sharing between P and Q in addition to that fixed by ϕ.

[7] We assume some form of non-ambiguous encoding of composed labels and output strings, but in our presentation we simply present these encoded values as tuples.

C Definitions

C.1 Cryptographic definitions

Message Authentication Codes SYNTAX. A message authentication code scheme Π is a triple of PPT algorithms $(\mathsf{Gen}, \mathsf{Auth}, \mathsf{Ver})$. On input 1^λ, where λ is the security parameter, the randomized key generation algorithm returns a fresh key. On input key and message m, the deterministic MAC algorithm Auth returns a tag t. On input key, m and t, the deterministic verification algorithm Ver returns T or F indicating whether t is a valid MAC for m relative to key. We require that, for all $\lambda \in \mathbb{N}$, all $\mathsf{key} \in [\mathsf{Gen}(1^\lambda)]$ and all m, it is the case that $\mathsf{Ver}(\mathsf{key}, \mathsf{m}, (\mathsf{Auth}(\mathsf{key}, \mathsf{m}))) = \mathsf{T}$.

SECURITY. We use the standard notion of existential unforgeability for MACs [6]. We say that Π is existentially unforgeable if $\mathsf{Adv}^{\mathsf{Auth}}_{\mathcal{A},\Pi}(\lambda)$ is negligible for every ppt adversary $\mathcal{A}$, where advantage is defined as the probability that the game in Figure 5 (top) returns T.

Digital Signature Schemes SYNTAX. A signature scheme Σ is a triple of PPT algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$. On input 1^λ, where λ is the security parameter, the randomized key generation algorithm returns a fresh key pair $(\mathsf{pk}, \mathsf{sk})$. On input secret key sk and message m, the possibly randomized signing algorithm Sign returns a signature σ. On input public key pk, m and σ, the deterministic verification algorithm Vrfy returns T or F indicating whether σ is a valid signature for m relative to pk. We require that, for all $\lambda \in \mathbb{N}$, all $(\mathsf{pk}, \mathsf{sk}) \in [\mathsf{Gen}(1^\lambda)]$ and all m, it is the case that $\mathsf{Vrfy}(\mathsf{pk}, \mathsf{m}, (\mathsf{Sign}(\mathsf{sk}, \mathsf{m}))) = \mathsf{T}$.

SECURITY. We use the standard notion of existential unforgeability for signature schemes [25]. We say that Σ is existentially unforgeable if $\mathsf{Adv}^{\mathsf{UF}}_{\mathcal{A},\Sigma}(\lambda)$ is negligible for every ppt adversary $\mathcal{A}$, where advantage is defined as the probability that the game in Figure 5 (bottom) returns T.

Authenticated Encryption Schemes SYNTAX. An authenticated encryption scheme Λ is a triple of PPT algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. On input 1^λ, where λ is the security parameter, the randomized key generation algorithm returns a fresh key. On input key key and message m, the randomized encryption algorithm Enc returns a ciphertext m'. On input key key and ciphertext m', the deterministic decryption algorithm Dec returns the decrypted message m, or $\perp$ if the ciphertext is found to be invalid. We require that, for all $\lambda \in \mathbb{N}$, all $\mathsf{key} \in [\mathsf{Gen}(1^\lambda)]$ and all m, it is the case that $\mathsf{m} = \mathsf{Dec}(\mathsf{key}, \mathsf{Enc}(\mathsf{key}, \mathsf{m}))$.

SECURITY. We use the standard notions of indistinguishability and existential unforgeability for authenticated encryption schemes [32]. We say that Λ provides ciphertext indistinguishability if $\mathsf{Adv}^{\mathsf{IND}}_{\mathcal{A},\Lambda}(\lambda)$ is negligible for every ppt adversary $\mathcal{A}$, where advantage is defined as the probability that the game in Figure 6 (top) returns T over the random guess. We say that Λ is existentially unforgeable if $\mathsf{Adv}^{\mathsf{UF}}_{\mathcal{A},\Lambda}(\lambda)$ is negligible for every ppt adversary $\mathcal{A}$, where advantage is defined as the probability that the game in Figure 6 (bottom) returns T.

```
Game $\mathsf{Auth}^{\Pi,\mathcal{A}}(1^\lambda)$:
    $\mathsf{List} \leftarrow []$
    $\mathsf{key} \leftarrow_\$ \mathsf{Gen}(1^\lambda)$
    $(\mathsf{m}, \mathsf{t}) \leftarrow_\$ \mathcal{A}^{\mathsf{Auth}}(1^\lambda)$
    Return $\mathsf{Ver}(\mathsf{key}, \mathsf{m}, \mathsf{t}) = \mathsf{T} \wedge m \notin \mathsf{List}$

Oracle $\mathsf{Auth}(\mathsf{m})$:
    $\mathsf{List} \leftarrow (\mathsf{m} : \mathsf{List})$
    $\mathsf{t} \leftarrow \mathsf{Mac}(\mathsf{key}, \mathsf{m})$
    Return $\mathsf{t}$
```

```
Game $\mathsf{UF}^{\Sigma,\mathcal{A}}(1^\lambda)$:
    $\mathsf{List} \leftarrow []$
    $(\mathsf{pk}, \mathsf{sk}) \leftarrow_\$ \mathsf{Gen}(1^\lambda)$
    $(\mathsf{m}, \sigma) \leftarrow_\$ \mathcal{A}^{\mathsf{Sign}}(1^\lambda, \mathsf{pk})$
    Return $\mathsf{Vrfy}(\mathsf{pk}, \mathsf{m}, \sigma) = \mathsf{T} \wedge m \notin \mathsf{List}$

Oracle $\mathsf{Sign}(\mathsf{m})$:
    $\mathsf{List} \leftarrow (\mathsf{m} : \mathsf{List})$
    $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, \mathsf{m})$
    Return $\sigma$
```

Fig. 5. Games defining the security of a MAC scheme (left) and a signature scheme (right).

```
Game $\mathsf{IND}^{\Lambda,\mathcal{A}}(1^\lambda)$:
    $\mathsf{List} \leftarrow []$
    $\mathsf{key} \leftarrow_\$ \mathsf{Gen}(1^\lambda)$
    $(\mathsf{m}_0, \mathsf{m}_1) \leftarrow_\$ \mathcal{A}_1^{\mathsf{Encrypt},\mathsf{Decrypt}}(1^\lambda)$
    $b \leftarrow_\$ \{0, 1\}$
    $\mathsf{m}' \leftarrow_\$ \mathsf{Enc}(\mathsf{key}, \mathsf{m}_b)$
    $b' \leftarrow_\$ \mathcal{A}_2^{\mathsf{Encrypt},\mathsf{Decrypt}}(\mathsf{m}_0, \mathsf{m}_1, \mathsf{m}')$
    If $\mathsf{m}' \in \mathsf{List}$: $b' \leftarrow_\$ \{0, 1\}$
    Return $b = b'$

Oracle $\mathsf{Encrypt}(\mathsf{m})$:
    Return $\mathsf{Enc}(\mathsf{key}, \mathsf{m})$

Oracle $\mathsf{Decrypt}(\mathsf{m}')$:
    $\mathsf{List} \leftarrow (\mathsf{m}' : \mathsf{List})$
    $\mathsf{m} \leftarrow \mathsf{Dec}(\mathsf{key}, \mathsf{m}')$
    Return $\mathsf{m}$
```

```
Game $\mathsf{UF}^{\Lambda,\mathcal{A}}(1^\lambda)$:
    $\mathsf{List} \leftarrow []$
    $\mathsf{key} \leftarrow_\$ \mathsf{Gen}(1^\lambda)$
    $\mathsf{m}' \leftarrow_\$ \mathcal{A}^{\mathsf{Encrypt}}(1^\lambda)$
    If $\mathsf{Dec}(\mathsf{key}, \mathsf{m}') \neq \perp \wedge \mathsf{m}' \notin \mathsf{List}$:
        Return $\mathsf{T}$
    Return $\mathsf{F}$

Oracle $\mathsf{Encrypt}(\mathsf{m})$:
    $\mathsf{m}' \leftarrow \mathsf{Sign}(\mathsf{sk}, \mathsf{m})$
    $\mathsf{List} \leftarrow (\mathsf{m}' : \mathsf{List})$
    Return $\mathsf{m}'$
```

Fig. 6. Games defining ciphertext indistinguishability (top) and existential unforgeability (bottom) of an authenticated encryption scheme.

C.2 Key Exchange

We recall here the notion of Key Exchange for Attested Computation (AttKE) from [3]. This notion is tailored to establish a secure key with a remote program running inside an IEE; the remote program includes a first stage where the key is derived using a key exchange subprogram called $\mathsf{Rem}_{\mathsf{KE}}$ and an arbitrary second stage program that uses the derived key. Active security in the key exchange protocol is achieved by running $\mathsf{Rem}_{\mathsf{KE}}$ using attestation mechanisms provided by the IEE. We will see in this paper that the exact same notion of AttKE can be used together with Labelled Attested Computation in a much wider program composition context and, particularly, to enable multiple parties to establish independent secure channels to the same IEE.

SYNTAX. An AttKE is defined as a pair of algorithms $(\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}})$. On input the security parameter 1^λ and the local party identity id, Setup generates the part of the key exchange intended for remote execution $\mathsf{Rem}_{\mathsf{KE}}$, and the initial state st_L of the local part of the key exchange $\mathsf{Loc}_{\mathsf{KE}}$. Setup is intended to generate a fresh instance of the AttKE protocol between the party with identity id and a remote IEE. The dynamically generated $\mathsf{Rem}_{\mathsf{KE}}$ will be run remotely in an IEE, under the protection of a LAC scheme, following some composition pattern. On input m and local state st_L, $\mathsf{Loc}_{\mathsf{KE}}$ returns the next message intended for the remote part of the key exchange and an updated state.

We require the local and remote parts of an AttKE to keep a set of variables in their states. The execution state of the AttKE is kept as $\delta \in \{\mathsf{derived}, \mathsf{accept}, \mathsf{reject}, \perp\}$.

The derived key is stored as key. It is supposed to be $\perp$ if $\delta \notin \{\mathsf{derived}, \mathsf{accept}\}$. The identity of the owner of the instance is represented as oid. This will be initialised on the fly for the remote part. The identity of the partner of the session is stored in variable pid. This will typically be set at generation for $\mathsf{Rem}_{\mathsf{KE}}$ and constructed during execution for $\mathsf{Loc}_{\mathsf{KE}}$. Finally the session identifier sid, will typically be constructed on the fly. An AttKE is correct if, after an honest run of a local instance and the corresponding remote instance, both accept, derive the same key and agree on sid, pid and oid.

SECURITY. The adversary model for AttKE security is tailored so that active security is provided when adding attestation to the remote part. This adversary is a middle ground between a passive and an active adversary. The security model considers an adversary which has access to oracles whose behaviour depend on a bit b and a list of pairs of real and fake keys, one for each instance. In addition to the oracles initialising new local and remote instances (multiple instances of the same remote instance can be created), the following oracles are provided:

- Reveal: when queried on a local or remote instance, it returns the corresponding derived key.
- Test: when queried on a local or remote instance, it returns $\perp$ if $\delta \neq \mathsf{accept}$; otherwise, if $b = 0$ it returns key and it returns $\mathsf{fake}(\mathsf{key})$ if $b = 1$.
- Send allows delivering messages between instances, making sure that messages from remote instances to local instances are reliably delivered. The messages delivered to remote instances, however, are arbitrary. This oracle updates the instances according to the message delivered and returns the response, together with the corresponding pid, sid and δ.

The model keeps track of local instances $\mathsf{Loc}^{l}_{\mathsf{KE}}$ created by the local identity id and remote instances $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$ (there can be many copies of $\mathsf{Rem}_{\mathsf{KE}}$ for each locally initialized session. A local and a remote instance are partnered if $\delta^{i,j}, \delta^{l} \in \{\mathsf{derived}, \mathsf{accept}\}$ and they agree on sid. We further restrict the adversary, by disallowing Test queries if Reveal was queried for this instance or a partnered instance.

We say that a protocol ensures valid partners if, for every partnered $\mathsf{Loc}^{l}_{\mathsf{KE}}$ and $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$, $\mathsf{pid}^{l} = \mathsf{oid}^{i,j}$, $\mathsf{oid}^{l} = \mathsf{pid}^{i,j}$ and the derived key is the same for both instances. We say that a protocol ensures confirmed partners if when an instance of the key exchange accepts, it has at least one partner. We say that a protocol ensure unique partners if each instance is partnered with at most one other instance. A protocol ensures two sided authentication if it ensures these three properties with overwhelming probability, in the presence of an adversary with access to the aforementioned oracles.

Definition 3 (AttKE security). *An AttKE protocol is secure if the protocol ensures two sided authentication, and for any ppt adversary $\mathcal{A}$, as described above, and for any local identity* id*, the probability of $\mathcal{A}$ guessing b is overwhelmingly close to* $1/2$.

D LAC correctness and minimal leakage

D.1 Correctness definition

Intuitively, a LAC scheme is correct if, for any given program P and attested label set L^*, assuming an honest execution of all components in the scheme, both locally and remotely, the local user is able to accurately reconstruct a partial view of the I/O sequence that took place in the remote environment, for an arbitrary set of labels L (which may or may not be related to L^*). For non-attested labels, i.e., labels in $L \setminus L^*$, we restrict the I/O behaviour of the compiled program inside the IEE imposing that it is identical to that of the original program. For this reason, the set of labels L should be seen as a parameter that can be used by higher level protocols relying on LAC to specify the *partial* local view that may interest a particular party interacting with a remote machine. Different parties may be interested in different partial views, including both attested an non-attested labels, and the protocol should be correct for all of them. More technically, suppose the compiled program is run under handle hdl^* in remote machine $\mathcal{M}$, with random coins $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)$ and on labelled input sequence $(l_1, i_1, \ldots, l_n, i_n)$. Suppose also that, running the original program on the same random coins and inputs yields

$$\mathsf{Trace}_{R[\mathsf{st};\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)]}(l_1, i_1, \ldots, l_n, i_n) = \\ (l_1, i_1, o_n, \ldots, l_n, i_n, o_n)$$

Then, for any set of labels L, if a local user recovers outputs $(o'_1, \ldots, o'_m)$ corresponding to labelled inputs (l_{k_1}, i_{k_1}) to (l_{k_m}, i_{k_m}), where $l_{k_j} \in L$, it must be the case that $(o'_1, \ldots, o'_m) = (o_{k_1}, \ldots, o_{k_m})$. Outputs for attested labels are passed through Attest and Verify, whereas inputs and outputs for non-attested labels are processed independently of these algorithms. The following definition formalizes the notion of a local user *correctly remotely executing program P* using labelled attested computation.

```
**Game** $\mathsf{Corr}_{\mathsf{LAC},\mathcal{A}}(1^\lambda)$:
$\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, L^*, L, n, \mathsf{st}_{\mathcal{A}}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$
$\mathsf{hdl}^* \leftarrow \mathcal{M}.\mathsf{Load}(P^*)$
$\mathsf{st}_V \leftarrow (\mathsf{prms}, P, L^*)$
For $k \in [1..n]$ :
    $(l_k, i_k, \mathsf{st}_{\mathcal{A}}) \leftarrow\$ \mathcal{A}_2(o^*_1, \ldots, o^*_{k-1}, \mathsf{st}_{\mathcal{A}})$
    If $l_k \in L \cap L^*$ :
        $o^*_k \leftarrow\$ \mathsf{Attest}^{\mathcal{M}}(\mathsf{prms}, \mathsf{hdl}^*, l_k, i_k)$
        $(o_k, \mathsf{st}_V) \leftarrow \mathsf{Verify}(\mathsf{prms}, l_k, i_k, o^*_k, \mathsf{st}_V)$
        If $o_k = \perp$ Then Return $\mathsf{F}$
    Else If $l_k \in L \setminus L^*$ :
        $o_k \leftarrow\$ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}^*, l_k, i_k)$
    Else Return $\mathsf{F}$
$T' \leftarrow \mathsf{filter}[L](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)]}(l_1, i_1, \ldots, l_n, i_n))$
$T \leftarrow \mathsf{filter}[L](l_1, i_1, o_1, \ldots, l_n, i_n, o_n)$ // get only labels in $L$
Return $T = T'$
```

Fig. 7. Game defining the correctness of LAC.

Definition 4 (Correctness). *A labelled attested computation scheme* LAC *is correct if, for all* λ *and all adversaries* $\mathcal{A}$*, the experiment in Fig. 7 always returns* T.

The adversary in this correctness experiment definition is choosing inputs, hoping to find a sequence that causes the attestation protocol to behave inconsistently with respect to the semantics of P (when these are made deterministic by hardwiring the same random coins used remotely). We use this approach to defining correctness because it makes explicit what is an honest execution of an attested computation scheme, when compared to the security experiment.

STRUCTURAL PRESERVATION. To simplify analysis of our constructions and proofs, we extend the correctness requirements on labelled attested computation schemes to preserve the structure of the input program when dealing with sequential composition, and to modify only the part of the code that will be attested. Formally, we impose that, for all global parameters, given any program $R = \langle P\,;\,Q\rangle_{\phi,p,q}$, and an attested label set L^* that contains only labels of the form (p, l), then there exists a (unique) compiled program P^*, such that, $\langle P^*\,;\,Q\rangle_{\phi,p,q} = \mathsf{Compile}(\mathsf{prms}, R, L)$. Note that this implies that the state of compiled program P^* somehow encodes the state of P in a way that is transparent for ϕ, and we will loosely rely on this when referring to the execution state of P^* and extracting values from it. Note also that, for composed programs compiled in this way, the unnatested I/O behaviour of the second program will be identical to that of the original program.

D.2 Minimal Leakage

The following definition imposes that nothing from the internal state of the source programs (in addition to what is public, i.e., the code and I/O sequence) is leaked in the trace of the compiled program.

Definition 5 (Minimal leakage). *A labelled attested computation scheme* LAC *ensures security with minimal leakage if it is secure according to Definition 1 and there exists a ppt simulator* $\mathcal{S}$ *that, for every adversary* $\mathcal{A}$*, the following distributions are identical:*

$$\{\,\mathsf{Leak\text{-}Real}_{\mathsf{LAC},\mathcal{A}}(1^\lambda)\,\} \approx \{\,\mathsf{Leak\text{-}Ideal}_{\mathsf{LAC},\mathcal{A},\mathcal{S}}(1^\lambda)\,\}$$

where games $\mathsf{Leak\text{-}Real}_{\mathsf{LAC},\mathcal{A}}$ *and* $\mathsf{Leak\text{-}Ideal}_{\mathsf{LAC},\mathcal{A},\mathcal{S}}$ *are shown in Fig. 8.*

Notice that we allow the simulator to replace the global parameters of the machine with some value prms for which it can keep some trapdoor information. Intuitively this means that one can construct a perfect simulation of the remote trace by simply appending cryptographic material to the local trace. This property is important when claiming that the security of a cryptographic primitive is preserved when it is run within an attested computation scheme (one can simply reduce the advantage of an adversary attacking the attested trace, to the security of the original scheme using the minimum leakage simulator).

```
Game Leak-Real_{LAC,A}(1^λ):
  PrgList ← []
  prms ←$ M.Init(1^λ)
  b ←$ A^O(prms)
  Return b

Oracle Run(hdl, l, i):
  Return M.Run(hdl, l, i)

Oracle Compile(P, L):
  P* ← Compile(prms, P, L)
  PrgList ← P* : PrgList
  Return P*

Oracle Load(P):
  Return M.Load(P)
```

```
Game Leak-Ideal_{LAC,A,S}(1^λ):
  PrgList ← []; List ← []
  hdl ← 0
  (prms, st_S) ←$ S_1(1^λ)
  b ←$ A^O(prms)
  Return b

Oracle Run(hdl, l, i):
  (P*, st) ← List[hdl]
  If (P*, L, P) ∈ PrgList :
    o ←$ P[st](l, i)
    (o*, st_S) ←$ S_2(hdl, P, L, l, i, o, st_S)
  Else:
    (o*, st_S) ←$ S_3(hdl, P*, l, i, st, st_S)
  List[hdl] ← (P*, st)
  Return o*

Oracle Compile(P, L):
  P* ← Compile(prms, P, L)
  PrgList ← (P*, L, P) : PrgList
  Return P*

Oracle Load(P*):
  hdl ← hdl + 1
  List[hdl] ← (P*, ϵ)
  Return hdl
```

Fig. 8. Games defining minimum leakage of LAC.

D.3 Correctness of our LAC scheme

It is easy to see that our LAC scheme is correct, provided that the underlying signature and message authentication schemes are correct, and that it preserves the structure of compiled programs. To see that this is the case, note that during the execution of P^* for $l_k \in L$, unless a MAC or signature verification fails, the I/O sequence provided by Verify will match the one reconstructed in T' (the inputs are the same, and the associated randomness tapes are fixed by $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)$), and therefore $T = T'$. Since these algorithms are only used for attested labels, we only need to consider this possibility for labels $l \in L^* \cap L$. Now, observe that if the message authentication code scheme is correct, then the MAC verification will never fail, and if the message signature scheme is correct, then the signature verification will never fail. This is the case because the combined operations of P^*, Attest, the signing IEE running S^* and the security module lead to tags and signatures on pairs (P^*, ios_l) that exactly match the inputs provided to the verification algorithms in $\Pi.\mathsf{Ver}$ and $\Sigma.\mathsf{Verify}$. This gives us that the received trace and the reconstructed trace will be the same for all labels in L.

E Proof of Theorem 1

The proof is a sequence of three games presented in Figure 9 and Figure 10. The first game is simply the LAC security game instantiated with our protocol.

In game $\mathsf{G_1}^{\mathsf{AC},\mathcal{A}}(1^\lambda)$, the adversary loses whenever a sforge event occurs. Intuitively, this event corresponds to the adversary producing a signature that was not computed by the signing process with handle 0, and hence constitutes a forgery with respect to Σ. Given that the two games are identical until this event occurs, we have that

$$\Pr[\,\mathsf{Att}^{\mathsf{LAC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G_1}^{\mathsf{LAC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \le \Pr[\,\mathsf{sforge}\,]\,.$$

```
Game G0_{LAC,A}(1^λ):
$\mathsf{prms} \leftarrow\!\!\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\!\!\$\ \mathcal{A}_1(\mathsf{prms})$

$P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$
For $k \in [1..n]$:
    $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\!\!\$\ \mathcal{A}_2^{\mathcal{M}}(\mathsf{st}_V, \mathsf{st}_\mathcal{A})$
    Parse $(o_k, \sigma) \leftarrow o_k^*$
    If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (P^*, \mathsf{filter}[l]((l, i_k, o_k) : \mathsf{ios})))$:
      $\mathsf{ios} \leftarrow ((l, i_k, o_k) : \mathsf{ios})$
    Else: Return F


$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For $\mathsf{hdl}^*$ s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
    $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$
    $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
    If $T \sqsubseteq T'$ Return F
Return T
```

```
Game G1_{LAC,A}(1^λ):
$\mathsf{prms} \leftarrow\!\!\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\!\!\$\ \mathcal{A}_1(\mathsf{prms})$
$\mathsf{sforge} \leftarrow \mathsf{F}$
$P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$
For $k \in [1..n]$:
    $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\!\!\$\ \mathcal{A}_2^{\mathcal{M}}(\mathsf{st}_V, \mathsf{st}_\mathcal{A})$
    Parse $(o_k, \sigma) \leftarrow o_k^*$
    If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (P^*, \mathsf{filter}[l]((l, i_k, o_k) : \mathsf{ios})))$:
      $\mathsf{ios} \leftarrow ((l, i_k, o_k) : \mathsf{ios})$
    Else: Return F
    If $((P^*, (l, i_1, o_1, \ldots, l, i_k, o_k), \star), \sigma') \notin \mathsf{Trace}_\mathcal{M}(0)$:
      $\mathsf{sforge} \leftarrow \mathsf{T}$; Return F
$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For $\mathsf{hdl}^*$ s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
    $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$
    $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
    If $T \sqsubseteq T'$ Return F
Return T
```

Fig. 9. First game hop for the proof of security of our AC protocol.

We upper bound the distance between these two games, by constructing an adversary $\mathcal{B}$ against the existential unforgeability of signature scheme Σ in S^* such that

$$\Pr[\,\mathsf{sforge}\,] \le \mathsf{Adv}^{\mathsf{UF}}_{\Sigma,\mathcal{B}}(\lambda)$$

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G_1}^{\mathsf{LAC},\mathcal{A}}$ as follows: the operation of machine $\mathcal{M}$ is simulated exactly with the caveat that the signing operations performed within the process loaded by the security module are replaced with calls to the Sign oracle provided in the existential unforgeability game. More precisely, whenever process 0 in the remote machine is expected to compute a signature on message m, algorithm $\mathcal{B}$ calls its own oracle on (P^*, m) to obtain σ.

When sforge is set, according to the rules of game $\mathsf{G_1}^{\mathsf{LAC},\mathcal{A}}$, algorithm $\mathcal{B}$ outputs message $(P^*, \mathsf{filter}[l](\mathsf{ios}))$ and candidate signature σ. It remains to show that this is a valid forgery. To see this, first observe that this is indeed a valid signature, as signature verification is performed on these values immediately before sforge occurs. It suffices to establish that message $(P^*, (l, i_1, o_1, \ldots, l, i_k, o_k))$ could not have been queried from the Sign oracle. Access to the signing key that allows signatures to be performed is only permitted to the special process with handle 0. From the construction of S^*, we know that producing such a signature would only occur via the inclusion of $(P^*, (l, i_1, o_1, \ldots, l, i_k, o_k))$ in its trace. Since we know that this is not the case, $(P^*, \mathsf{filter}[l](\mathsf{ios}))$ could not have been queried from the signature oracle. We conclude therefore that $\mathcal{B}$ outputs a valid forgery whenever sforge occurs.

In game $\mathsf{G_2}^{\mathsf{LAC},\mathcal{A}}(1^\lambda)$, the adversary loses whenever a mforge event occurs. Intuitively, this event corresponds to the adversary producing a tag that was not computed by the security module, and hence constitutes a forgery with respect to Π. Given that the two games are identical until this event occurs, we have that

$$\Pr[\,\mathsf{G_1}^{\mathsf{LAC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G_2}^{\mathsf{LAC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{mforge}\,]\,.$$

```
Game G1_{LAC,A}(1^λ):
$\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$\mathsf{sforge} \leftarrow \mathsf{F}$
$P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$
For $k \in [1..n]$:
    $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_2^\mathcal{M}(\mathsf{st}_V, \mathsf{st}_\mathcal{A})$
    Parse $(o_k, \sigma) \leftarrow o_k^*$
    If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (P^*, \mathsf{filter}[l]((l, i_k, o_k) : \mathsf{ios})))$:
      $\mathsf{ios} \leftarrow ((l, i_k, o_k) : \mathsf{ios})$
    Else: Return F
    If $((P^*, (l, i_1, o_1, \ldots, l, i_k, o_k), \star), \sigma') \notin \mathsf{Trace}_\mathcal{M}(0)$:
      $\mathsf{sforge} \leftarrow \mathsf{T}$; Return F
$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For $\mathsf{hdl}^*$ s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
    $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$
    $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
    If $T \sqsubseteq T'$ Return F
Return T
```

```
Game G2_{LAC,A}(1^λ):
$\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$\mathsf{sforge} \leftarrow \mathsf{F}$; $\mathsf{mforge} \leftarrow \mathsf{F}$
$P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$
For $k \in [1..n]$:
    $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_2^\mathcal{M}(\mathsf{st}_V, \mathsf{st}_\mathcal{A})$
    Parse $(o_k, \sigma) \leftarrow o_k^*$
    If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (P^*, \mathsf{filter}[l]((l, i_k, o_k) : \mathsf{ios})))$:
      $\mathsf{ios} \leftarrow ((l, i_k, o_k) : \mathsf{ios})$
    Else: Return F
    If $((P^*, (l, i_1, o_1, \ldots, l, i_k, o_k), \star), \sigma') \notin \mathsf{Trace}_\mathcal{M}(0)$:
      $\mathsf{sforge} \leftarrow \mathsf{T}$; Return F
    If $\nexists\, \mathsf{hdl}^*.\ \mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^* \wedge$
    $(l, i_1, o_1, \ldots, l, i_k, o_k) \sqsubseteq \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(\mathsf{Trace}_\mathcal{M}(\mathsf{hdl}^*))$:
      Then $\mathsf{mforge} \leftarrow \mathsf{T}$; Return F
$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For $\mathsf{hdl}^*$ s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
    $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$
    $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
    If $T \sqsubseteq T'$ Return F
Return T
```

Fig. 10. Second game hop for the proof of security of our AC protocol.

We upper bound the distance between these two games, by constructing an adversary $\mathcal{C}$ against the existential unforgeability of MAC scheme Π in the security module such that

$$\Pr[\,\mathsf{mforge}\,] \leq \mathsf{Adv}^{\mathsf{Auth}}_{\Pi,\mathcal{C}}(\lambda)$$

Adversary $\mathcal{C}$ simulates the environment of $\mathsf{G_2}^{\mathsf{LAC},\mathcal{A}}$ as follows: the operation of machine $\mathcal{M}$ is simulated exactly with the caveat that the MAC operations computed inside the internal security module are replaced with calls to the Auth oracle provided in the existential unforgeability game. More precisely, whenever a process running code R^* within an IEE in the remote machine requests a MAC on message m from the security module, algorithm $\mathcal{C}$ calls its own oracle on (P^*, m) to obtain t.

Let $T \leftarrow (l, i_1, o_1, \ldots, l, i_k, o_k)$. When mforge is set according to the rules of game $\mathsf{G_2}^{\mathsf{LAC},\mathcal{A}}$, algorithm $\mathcal{C}$ retrieves the trace of the process with handle 0 running S^*, locates the input/output pair $(((P^*, T), \mathsf{t}), \sigma')$ and outputs message (P^*, T) and candidate tag t. To see this is a valid forgery, first observe that, having failed the sforge check, we know that $(((P^*, T), \mathsf{t}), \sigma')$ is in the trace of the process with handle 0, so by its construction we also know that the corresponding input $((P^*, T), \mathsf{t})$ must contain a valid tag. It suffices to establish that message (P^*, T) could not have been queried from the Auth oracle. Suppose that the first part of the mforge check failed, i.e., that $\nexists\,\mathsf{hdl}^*.\ \mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}^*) = P^*$. Then, because the security module signs the code of the processes requesting the signatures, we are sure that such a query was never placed to the Auth oracle. Furthermore, any MAC query for a message starting with P^* must have been caused by the execution of an instance of P^*. Now suppose some instances of P^* were indeed running in the remote machine, but that none of them displayed the property $(l, i_1, o_1, \ldots, l, i_k, o_k) \sqsubseteq \mathsf{filter}[l](\mathsf{Translate}(\mathsf{ATrace}_{\mathcal{M}}(\mathsf{hdl}^*)))$. Then, by the construction of P^*, we can also exclude that (P^*, T) was queried from the MAC oracle. As such, we conclude that $\mathcal{C}$ outputs a valid forgery whenever mforge occurs.

To complete the proof, we argue that the adversary never wins in game $\mathsf{G_2}^{\mathsf{LAC},\mathcal{A}}$. To see this, observe that when the game reaches the final check, we have the guarantee that

$$\exists\,\mathsf{hdl}^*.\ \mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}^*) = P^* \wedge$$
$$(l, i_1, o_1, \ldots, l, i_n, o_n) \sqsubseteq \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)]}(\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl}^*))$$

Which exactly matches the final criteria of $T \sqsubseteq T'$.

To finish the proof, we must now show that this scheme also provides security with minimum leakage. This implies defining a ppt simulator $\mathcal{S}$ that provides identical distributions with respect to experiment in Figure 8. This is easy to ascertain given the simulator behaviour described in Figure 11: $\mathcal{S}_1$ and $\mathcal{S}_3$ follow the exact description of the actual machine, modulo the generation of $(\mathsf{pk}, \mathsf{sk})$ and key. $\mathcal{S}_2$ takes an external output produced by $P[\mathsf{st}](l, i)$ and returns an output in accordance to the behaviour of $\mathcal{M}$, which given our language $\mathcal{L}$ may differ from a real output only by the random coins. As such, the distribution provided by the simulator is indistinguishable to the one provided by a real machine, and our claim follows.

□

F MPC definitions

F.1 MPC correctness

The following definition formalizes the notion of n users *correctly running a function evaluation protocol* π for F.

Definition 6. *We say π is correct for functionality $\mathcal{F}$ on m inputs in r rounds if, for all λ, and all adversaries $\mathcal{A}$, the experiment in Figure 12 always returns* T.

DISCUSSION. Our correctness definition considers an honest execution environment, but includes a correctness adversary that is in charge of finding problematic inputs for the protocol and potentially erroneous execution schedules. It is parametrized by a number of inputs m and a number of rounds r.

The first stage of the experiment executes the Setup and Init algorithms that initialise both the remote machine and the parties' local states, and collects the public parameters for all of these participants (which we assume to be authenticated through

Simulator $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$

Simulator $\mathcal{S}$ will perform according to the $\mathcal{M}$ execution description.

– Upon input 1^λ, $\mathcal{S}_1$ generates a key pair for process S^*, a MAC key for the security module and initializes the traces and states as an empty list. The public key will be the public parameters, while the secret key be stored in its initial state.

```
$\underline{\mathcal{S}_1(1^\lambda)}$:
$\mathsf{key} \leftarrow\!\$\ \Pi.\mathsf{Gen}(1^\lambda)$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\$\ \Sigma.\mathsf{Gen}(1^\lambda)$
$\mathsf{Traces} \leftarrow [\,]$
Return $(\mathsf{pk}, (\mathsf{key}, \mathsf{sk}, \mathsf{Traces}))$
```

– $\mathcal{S}_2$ maintains a list of traces Traces with the respective list ios and stage stage. Given this, it masks output o^* as if produced by an actual machine execution.

```
$\underline{\mathcal{S}_2(\mathsf{hdl}, P, L, l, i, o^*, \mathsf{st}_\mathcal{S})}$:
Parse $(\mathsf{key}, \mathsf{sk}, \mathsf{Traces}) \leftarrow \mathsf{st}_\mathcal{S}$
If $\mathsf{Traces}[\mathsf{hdl}] \neq [\,]$: $\mathsf{ios} \leftarrow \mathsf{Traces}[\mathsf{hdl}]$
Else: $\mathsf{ios} \leftarrow [\,]$
If $l \in L^*$:
    $\mathsf{ios} \leftarrow (l, i, o^*) : \mathsf{ios}$
    $\mathsf{m} \leftarrow (o^*, \Pi.\mathsf{Mac}(\mathsf{key}, P, \mathsf{filter}[l](\mathsf{ios})))$
Else: $\mathsf{m} \leftarrow o^*$
$\mathsf{Traces}[\mathsf{hdl}] \leftarrow \mathsf{ios}$; $\mathsf{st}_\mathcal{S} \leftarrow (\mathsf{key}, \mathsf{sk}, \mathsf{Traces})$
Return $(\mathsf{m}, \mathsf{st}_\mathcal{S})$
```

– $\mathcal{S}_3$ standardly computes the next output given input i, program R and state st. $\mathsf{Sim}(P^*[\mathsf{st}], l, i, \mathsf{ios}, \mathsf{key})$ refers to the operation of executing P^* with label l and input i for trace ios and considering a security module with key key, producing output m and updated trace ios. The result is afterwards treated similar to $\mathcal{S}_2$.

```
$\underline{\mathcal{S}_3(\mathsf{hdl}, P^*, l, i, \mathsf{st}, \mathsf{st}_\mathcal{S})}$:
Parse $(\mathsf{key}, \mathsf{sk}, \mathsf{Traces}) \leftarrow \mathsf{st}_\mathcal{S}$
If $\mathsf{hdl} = 0$:
    Parse $(\mathsf{m}, \mathsf{t}) \leftarrow i^*$
    If $\Pi.\mathsf{Ver}(\mathsf{key}, \mathsf{t}, \mathsf{m})$: Return $\Sigma.\mathsf{Sign}(\mathsf{sk}, \mathsf{m})$
    Else Return $\perp$
If $\mathsf{Traces}[\mathsf{hdl}] \neq [\,]$: $\mathsf{ios} \leftarrow \mathsf{Traces}[\mathsf{hdl}]$
Else: $\mathsf{ios} \leftarrow [\,]$
$L \leftarrow \mathsf{labels}(P^*)$
$(\mathsf{m}, \mathsf{ios}) \leftarrow\!\$\ \mathsf{Sim}(P^*[\mathsf{st}], l, i, \mathsf{ios}, \mathsf{key})$
$\mathsf{Traces}[\mathsf{hdl}] \leftarrow \mathsf{ios}$; $\mathsf{st}_\mathcal{S} \leftarrow (\mathsf{key}, \mathsf{sk}, \mathsf{Traces})$
Return $(\mathsf{m}, \mathsf{st}_\mathcal{S})$
```

Fig. 11. Description of simulator $\mathcal{S}$

the paper, e.g., using a PKI). In the second part of the experiment, the adversary chooses a sequence of m inputs for the functionality, interleaving different parties in an arbitrary way. The sequence $(\mathsf{id}_1, \ldots, \mathsf{id}_m)$ implicitly defines a schedule for the execution of our protocol, which should ensure that the inputs of each party are provided to the functionality in precisely this order.

The last stage of the correctness experiment emulates the protocol execution, alternating between local steps and remote steps. The Remote algorithm commands the scheduling of message exchanges; this algorithm is always invoked first and its output indicates the next party to be activated, the message this party will receive, and whether or not the party is expected to provide an input. The protocol is run for r rounds, at which point its outputs are retrieved via Output. The adversary wins the game if it can force the game to produce a set of outputs that wouldn't be obtained by simply running the functionality F with the given inputs in the provided order.

```
Game $\mathsf{Corr}_{\mathcal{F},\pi,r,m,\mathcal{A},\mathcal{M}}(1^\lambda)$:
/Trusted setup of machine and parties
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$\mathsf{prms} \leftarrow\!\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
For id $\in [1..n]$:
    $(\mathsf{st}_{\mathsf{id}}, \mathsf{pub}_{\mathsf{id}}) \leftarrow\!\$\ \mathsf{Setup}(\mathsf{prms}, \mathsf{id})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_n)$
For id $\in [1..n]$:
    $\mathsf{st}_{\mathsf{id}} \leftarrow\!\$\ \mathsf{Init}(\mathsf{st}_{\mathsf{id}}, \mathsf{Pub})$
    $\mathsf{out}_{\mathsf{id}_i} \leftarrow \epsilon$

/Adversarially scheduled ideal execution
$\mathsf{st}_{\mathsf{F}} \leftarrow \epsilon; \mathsf{st}_{\mathcal{A}} \leftarrow \epsilon$
For $i \in [1..m]$:
    $(\mathsf{id}_i, \mathsf{in}_i, \mathsf{st}_{\mathcal{A}}) \leftarrow\!\$\ \mathcal{A}(\mathsf{prms}, \mathsf{Pub}, \mathsf{st}_{\mathcal{A}})$
    $\mathsf{out}_{\mathsf{id}_i} \leftarrow \mathsf{out}_{\mathsf{id}_i} \,||\, \mathsf{F}[\mathsf{st}_{\mathsf{F}}](\mathsf{id}_i, \mathsf{in}_i)$

/Protocol execution
$F^* \leftarrow \mathsf{Compile}(\mathsf{prms}, \mathcal{F}, \mathsf{Pub})$
$\mathsf{hdl} \leftarrow \mathcal{M}.\mathsf{Load}(F^*)$
$\mathsf{t} \leftarrow \mathsf{T}; \mathsf{m} \leftarrow \epsilon; j \leftarrow 0$
$\mathsf{st}_R \leftarrow (\mathsf{id}_1, \ldots, \mathsf{id}_m)$/input schedule
For $i \in [1..r]$:
    If t:/Remote step
        $(\mathsf{id}, \mathsf{inreq}, \mathsf{m}', \mathsf{st}_R) \leftarrow\!\$\ \mathsf{Remote}^{\mathcal{M}}(\mathsf{prms}, \mathsf{hdl}, \mathsf{m}, \mathsf{st}_R)$
    Else:/Local step
        If $\mathsf{inreq} = \mathsf{T}$:
            If $\mathsf{id} \neq \mathsf{id}_j$ Return F
            $\mathsf{st}_{\mathsf{id}} \leftarrow\!\$\ \mathsf{AddInput}(\mathsf{in}_j, \mathsf{st}_{\mathsf{id}})$
            $j \leftarrow j + 1$
        $(\mathsf{st}_{\mathsf{id}}, \mathsf{m}) \leftarrow\!\$\ \mathsf{Process}(\mathsf{st}_{\mathsf{id}}, \mathsf{m}')$
    $\mathsf{t} \leftarrow \neg\mathsf{t}$
For id $\in [1..n]$:
    $\mathsf{out}'_1 \leftarrow \mathsf{Output}(\mathsf{st}_{\mathsf{id}})$
Return $(\mathsf{out}_1, \ldots, \mathsf{out}_n) = (\mathsf{out}'_1, \ldots, \mathsf{out}'_n)$
```

Fig. 12. Game defining protocol correctness.

REMARK. The correctness experiment shows the crucial scheduling role of the Remote algorithm, which is run in an untrusted environment in the remote machine. Here we deviate from the standard approach in the UC setting, where the simulation-based definition of security is taken as providing sufficient detail to evaluate correctness of the protocol. Indeed, as other simulation-based definitions, our security experiment below will impose some input/output consistency conditions on the protocol. Intuitively, these must hold for any adversarially chosen Remote scheduling algorithm, as the adversary has full control of the remote machine and scheduling can be arbitrarily controlled by the attacker. However, we believe that there is added value in including a separate correctness definition, where the scheduling tasks of the non-security critical parts of the protocol can be specified as a first class feature of the protocol syntax. This also clarifies the envisioned execution model and makes it explicit that untrusted code running in an adversarially run machine is only relevant for correctness purposes.

G MPC protocol definitions

G.1 AttKE and utility

We define utility almost as in the AC case from [3]. The main difference is the composition context we allow for. In [3] the composition context is restricted to a key exchange composed sequentially with another program. Here we allow for the key exchange to be composed in parallel with other arbitrary programs and then sequentially with another program. The proof follows the same lines. The utility security experiment intuitively states that the adversary cannot distinguish between a derived key and a random key, whenever the key exchange has been performed between an honest party and a remote machine running $\mathsf{Rem}_{\mathsf{KE}}$ within an IEE, and $\mathsf{Rem}_{\mathsf{KE}}$ is composed with other arbitrary programs as described above. The reason this parallel/sequential composition pattern does not harm security is that the parts of the state belonging to different parallel-composed programs are disjoint, and sequential composition only reveals controlled information to other programs. Indeed, as in [3], we restrict sequential composition in the utility theorem to pass only specific parts of the state of the key exchange program to the following phase: mapping function ϕ_{key} passes on the derived key, the session and party identifiers, and the state ($\mathsf{derived}$ or accept) of the key exchange. Contrary to [3] this is not enough to define our mapping function, as other programs composed in parallel with the remote key exchange need to pass states to the next phase as well. To that extent, if $\phi_1, \ldots, \phi_n$ are mapping functions, we define $\phi_1^{l_1}|\ldots|\phi_n^{l_n}$ as $\phi^* := \mathsf{st}.l_i \mapsto \phi_i(\mathsf{st}.l_i)$. If the state comes from the program $\langle P_1|\ldots|P_n\rangle_{(l_1,\ldots,l_n)}$, this mapping function maps the state belonging to each P_i using ϕ_i. In our composition context, we take the ϕ_i corresponding to the key exchange to be ϕ_{key}. This ensures that only the key is transmitted to the following stage of the protocol, and not information supposed to be local to the key exchange protocol and not intended for further use.

In the experiment in Figure 13 the adversary has to distinguish between an ideal machine and a real world machine where an AttKE is run in parallel with other programs in the first phase of a LAC-compiled protocol. The machine $\mathcal{M}$ represents the remote machine expected by the LAC protocol and the machine $\mathcal{M}'$ is a modification of machine $\mathcal{M}$ in which the key derived by a key-exchange session is magically replaced by a fresh key. In order to maintain consistency between the tested keys and the keys used in $\mathcal{M}'$, oracle $\mathcal{M}'.\mathsf{Run}$ takes two additional parameters: a list fake of pairs of keys and a flag tweak. If the flag is activated, the following modifications in the behaviour of $\mathcal{M}$ occur in $\mathcal{M}'$:

- It expects the sub-program being activated due to input label l to be a key exchange $\mathsf{Rem}_{\mathsf{KE}}$ instance. After running its transition function, $\mathcal{M}'$ checks if it has reached the $\mathsf{derived}$ or accept state. If so, it retrieves the derived key and if there is no association $(\mathsf{key}, \_)$ in fake it generates a fresh key key^* and appends $(\mathsf{key}, \mathsf{key}^*)$ to fake.
- Furthermore, if the key exchange process has entered accept state, it performs $\mathsf{st}.l.\mathsf{key} \leftarrow \mathsf{fake}(\mathsf{key})$, i.e., it replaces the derived key with a fake random one. Note that this will cause the fake key to be passed to the next stage of the sequentially composed program.

The oracles provided to the adversary provide access to the remote machine. Additionally the adversary can create new sessions of the key exchange using the NewSession oracle, where the remote key exchange is composed in parallel (with label l^*) with programs $P_1, \ldots, P_n$, followed with Q and compiled for LAC. Note that, given the structure of our parallel composition, the position in which a program is listed in the composition expression is irrelevant. The adversary makes the local part of the key exchange progress by using the Send oracle, provided that the message passes the LAC verification step for the relevant label. Finally the adversary can challenge a session by executing the Test oracle, which return either the real key of a fake key according to b (provided that the key exchange has reached a derived or accept state).

Theorem 3 (Local AttKE utility). *If the AttKE is correct and secure, and the LAC protocol is correct, secure and ensures minimal leakage, then for all ppt adversaries in the labelled utility experiment: the probability that the adversary violates the AttKE two-sided entity authentication is negligible; and the key secrecy advantage* $2 \cdot \Pr[\mathsf{guess}] - 1$ *is negligible.*

G.2 Details of the protocol

Details of the protocol can be found in Figure 14. The (untrusted) scheduling algorithm is shown in Figure 15. It is in charge of dispatching messages to/from the remote machine IEE using the Attest algorithm provided by the LAC, and animating the protocol to generate a correct execution for an arbitrary sequence of input-party interactions provided externally as an input schedule which is stored in its initial state. During the bootstrap stage, the Remote procedure interacts with one party at a time,[8] moving from one party to the next when the previous party has moved to its second stage. When all parties have completed the key exchange, the Remote procedure detects this in the output of the IEE (consistently with the properties of our sequential composition), and moves to the functionality execution stage.

In this second stage, the algorithm simply follows the provided input schedule. Moving to the next input is triggered by feeding the algorithm with an empty input provided by the previous party (this is syntactic book-keeping to match our correctness requirement, and it signals the fact that the previous output was correctly delivered to the previous party). The consequence of such an action is that Remote signals that a new input should be requested from the next party in the schedule. When an actual input is received, this is passed into the IEE using an unattested label of the form (q, id). The output is sent back to the same party.

H Proof of the Utility Theorem

In [4] the proof of utility, presented in Appendix C consists in a sequence of 4 games $\mathsf{G0}_{\mathsf{AttKE},\mathcal{A}}$ to $\mathsf{G3}_{\mathsf{AttKE},\mathcal{A}}$. The first hop removes the possibility of an AC forgery, thus

[8] Other options were of course possible for implementing Remote, and the core of our protocol is actually compatible with a totally asynchronous scheduling. Dealing with such issues is out of the scope of this paper.

```
**Game** $\mathsf{Att}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda, \mathsf{id})$:
$\mathsf{prms}_0 \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$\mathsf{prms}_1 \leftarrow\$ \mathcal{M}'.\mathsf{Init}(1^\lambda)$
$\mathsf{PrgList} \leftarrow [\,]$
$\mathsf{fake} \leftarrow [\,]$
$i \leftarrow 0$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(\mathsf{prms}_b)$
Return $b = b'$

**Oracle** $\mathsf{Load}(R^*)$:
$\mathsf{hdl}_0 \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
$\mathsf{hdl}_1 \leftarrow \mathcal{M}'.\mathsf{Load}(R^*)$
Return $\mathsf{hdl}_b$

**Oracle** $\mathsf{Run}(\mathsf{hdl}, l, x)$:
$o_0 \leftarrow\$ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, l, x)$
$\mathsf{tweak} \leftarrow \mathsf{F}$
If $(\mathsf{Program}_{\mathcal{M}'}(\mathsf{hdl}), l) \in \mathsf{PrgList}$ then $\mathsf{tweak} \leftarrow \mathsf{T}$
$(o_1, \mathsf{fake}) \leftarrow\$ \mathcal{M}'.\mathsf{Run}(\mathsf{hdl}, l, x, \mathsf{tweak}, \mathsf{fake})$
Return $o_b$

**Oracle** $\mathsf{Test}(i)$:
If $\mathsf{st}^i_{\mathsf{KE}}.\delta \neq \mathsf{accept}$: Return $\perp$
If $b = 0$: Return $\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key}$
Else: Return $\mathsf{fake}(\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key})$

**Oracle** $\mathsf{NewSession}(P_1, l_1, \phi_1, \ldots, P_n, l_n, \phi_n, l^*, Q, L^*)$:
If $\exists j, k$ such that $j \neq k \wedge l_j = l_k$: Return $\perp$
If $(p, (l^*, \epsilon)) \notin L^*$: Return $\perp$
$i \leftarrow i + 1$
$l^*_i \leftarrow (p, (l^*, \epsilon))$
$(\mathsf{st}^i_{\mathsf{KE}}, \mathsf{Rem}^i_{\mathsf{KE}}) \leftarrow\$ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \epsilon$
$\mathsf{RemComp} := \langle \mathsf{Rem}^i_{\mathsf{KE}} | P_1 | \ldots | P_n \rangle_{(l^*, l_1, \ldots, l_n)}$
$\phi^* := \phi^{l^*}_{\mathsf{key}} | \phi^{l_1}_1 | \ldots | \phi^{l_n}_n$
$R_i := \langle\, \mathsf{RemComp}\, ;\, Q\, \rangle_{\phi, p, q}$
$R^*_i \leftarrow\$ \mathsf{LAC.Compile}(\mathsf{prms}_b, R_i, L^*)$
$\mathsf{st}^i_V \leftarrow (R^*_i, L^*)$
$\mathsf{PrgList} \leftarrow (R^*_i, l^*_i) : \mathsf{PrgList}$
Return $R^*_i$

**Oracle** $\mathsf{Send}(o^*, i)$:
$o \leftarrow \mathsf{LAC.Verify}[\mathsf{st}^i_V](\mathsf{prms}, l^*_i, \mathsf{in}^i_{\mathsf{last}}, o^*)$
If $o = \perp$: Return $\perp$
$\mathsf{m}^* \leftarrow\$ \mathsf{Loc}_{\mathsf{KE}}[\mathsf{st}^i_{\mathsf{KE}}](o)$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \mathsf{m}^*$
If $\mathsf{st}^i_{\mathsf{KE}}.\delta \in \{\mathsf{derived}, \mathsf{accept}\} \wedge \mathsf{st}^i_{\mathsf{KE}}.\mathsf{key} \notin \mathsf{fake}$:
    $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return $\mathsf{m}^*$
```

Fig. 13. Utility of adversarially composed AttKE.

ensuring that messages are delivered properly from the remote program executing the key exchange to the local party. The second hop, using minimal leakage, replaces execution in the remote machine by a simulated execution based on a real execution of the compiled program. The final game hop replaces the execution of the remote key exchange by call to the AttKE security oracles, and conclude immediately by security

```
**algorithm** $\mathsf{Setup}(\mathsf{prms}, \mathsf{id})$:
$\mathsf{st_{id}.id} \leftarrow \mathsf{id}$; $\mathsf{st_{id}.prms} \leftarrow \mathsf{prms}$
$(\mathsf{st}_L, \mathsf{Rem_{KE}}) \leftarrow \mathsf{Setup_{KE}}(1^\lambda, \mathsf{id})$
$\mathsf{st_{id}.st}_L \leftarrow \mathsf{st}_L$; $\mathsf{st_{id}.pub} \leftarrow \mathsf{Rem_{KE}}$
Return $(\mathsf{st_{id}}, \mathsf{st_{id}.pub})$

**algorithm** $\mathsf{Compile}(\mathsf{prms}, \mathcal{F}, \mathsf{Pub})$:
$(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
$P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n}\ ;\ \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle\, \rangle_{\phi_{\mathsf{key}}, p, q}$
$L^* \leftarrow \{(p, (1, \epsilon)), \ldots, \{(p, (n, \epsilon))\}$ /Labels for $\mathsf{Rem}^i_{\mathsf{KE}}$ are empty
$P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, P, L^*)$
Return $P^*$

**algorithm** $\mathsf{Init}(\mathsf{st_{id}}, \mathsf{Pub})$:
$\mathsf{st_{id}.InList} \leftarrow [\,]$; $\mathsf{st_{id}.stage} \leftarrow 0$;
$\mathsf{st_{id}.seq_{in}} \leftarrow 0$; $\mathsf{st_{id}.seq_{out}} \leftarrow 1$; $\mathsf{st_{id}.in_{last}} \leftarrow \epsilon$
If $\mathsf{Pub}[\mathsf{st_{id}.id}] \neq \mathsf{st_{id}.pub}$ : Return $\perp$
$(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
$P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n}\ ;\ \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle\, \rangle_{\phi_{\mathsf{key}}, p, q}$
$L \leftarrow \{(p, (\mathsf{st_{id}.id}, \epsilon)), (q, \mathsf{st_{id}.id})\}$
$\mathsf{st_{id}.st}_V \leftarrow (P, L)$
Return $\mathsf{st_{id}}$

**algorithm** $\mathsf{Process}(\mathsf{st_{id}}, \mathsf{m})$:
/Bootstrap (attested labels)
if $\mathsf{st_{id}.stage} = 0$ :
  $(i, \mathsf{st_{id}.st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st_{id}.prms}, (p, (\mathsf{st_{id}.id}, \epsilon)), \mathsf{in_{last}}, \mathsf{m}, \mathsf{st}_V)$
  If $i = \perp$: Return $\perp$
  $(o, \mathsf{st_{id}.st}_L) \leftarrow\$\ \mathsf{Loc_{KE}}(\mathsf{st_{id}.st}_L, i)$
  $\mathsf{st_{id}.in_{last}} \leftarrow o$
  If $(\mathsf{st_{id}.st}_L.\mathsf{st_{KE}}.\delta) = \mathsf{accept}$ : Then stage $\leftarrow 1$
  $\mathsf{m}' \leftarrow (\mathsf{st_{id}.stage}, \mathsf{st_{id}.id}, o)$
  Return $(\mathsf{st_{id}}, \mathsf{m}')$

/Execution (non-attested labels)
if $\mathsf{st_{id}.stage} = 1$ :
  If $\mathsf{m} = \epsilon$ :/Input requested (empty message signal)
    $\mathsf{in} \leftarrow \mathsf{st_{id}.InList}[0]$
    $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st_{id}.ListIn_{id}}$
    $\mathsf{st_{id}.ListIn_{id}} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_{k\text{-}1})$
    $o \leftarrow\$\ \Lambda.\mathsf{Enc}(\mathsf{st_{id}.st}_L.\mathsf{key}, (\mathsf{st_{id}.seq_{in}}, \mathsf{in}))$
    $\mathsf{st_{id}.in_{last}} \leftarrow o$
    $\mathsf{st_{id}.seq_{in}} \leftarrow \mathsf{st_{id}.seq_{in}} + 2$
    $\mathsf{m}' \leftarrow (\mathsf{st_{id}.stage}, \mathsf{st_{id}.id}, o)$
    Return $(\mathsf{st_{id}}, \mathsf{m}')$
  Else:/Process received output
    $\mathsf{m}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{st_{id}.st}_L.\mathsf{key}, \mathsf{m})$
    If $\mathsf{m}' = (\mathsf{st_{id}.seq_{out}}, \mathsf{out}')$ :
      $\mathsf{st_{id}.seq_{out}} \leftarrow \mathsf{st_{id}.seq_{out}} + 2$
      $\mathsf{st_{id}.out} \leftarrow \mathsf{out}'$
      $\mathsf{m}' \leftarrow (\mathsf{st_{id}.stage}, \mathsf{st_{id}.id}, \epsilon)$
      Return $(\mathsf{st_{id}}, \mathsf{m}')$
    Else: Return $\perp$

**algorithm** $\mathsf{AddInput}(\mathsf{in}, \mathsf{st_{id}})$:
$\mathsf{st_{id}.InList} \leftarrow \mathsf{st_{id}.InList} + [\mathsf{in}]$
Return $\mathsf{st_{id}}$

**algorithm** $\mathsf{Output}(\mathsf{st_{id}})$:
Return $\mathsf{st_{id}.out}$
```

Fig. 14. General SMPC protocol.

```
**algorithm** $\mathsf{Remote}^{\mathcal{M}}(\mathsf{prms}, \mathsf{hdl}, \mathsf{m}, \mathsf{st}_R)$:
/Initial message
If $\mathsf{m} = \epsilon$ :
    $\mathsf{m} \leftarrow (0, 1, \epsilon)$/Force bootstrap start
    $\mathsf{st}_R.\mathsf{IdList} \leftarrow \mathsf{st}_R$/Input schedule
    $\mathsf{st}_R.\mathsf{stage} \leftarrow 0$

/Bootstrap (attested labels)
If $\mathsf{st}_R.\mathsf{stage} = 0$ :
    $(\mathsf{stage}_{\mathsf{id}}, \mathsf{id}, i) \leftarrow \mathsf{m}$
    $o \leftarrow \mathsf{LAC.Attest}^{\mathcal{M}}(\mathsf{prms}, \mathsf{hdl}, (p, (\mathsf{id}, \epsilon)), i)$
    If $o.\mathsf{stage} = 1$ :/IEE just finished bootstrap
      $\mathsf{st}_R.\mathsf{stage} = 1$; $\mathsf{inreq} \leftarrow \mathsf{T}$; $\mathsf{m}' \leftarrow \epsilon$
      $(\mathsf{id}_1, \ldots, \mathsf{id}_k) \leftarrow \mathsf{st}_R.\mathsf{IdList}$; $\mathsf{id} = \mathsf{id}_1$
      $\mathsf{st}_R.\mathsf{IdList} \leftarrow (\mathsf{id}_2, \ldots, \mathsf{id}_k)$
      Return $(\mathsf{id}, \mathsf{inreq}, \mathsf{m}', \mathsf{st}_R)$
    Else:/Just continue bootstrap
      If $\mathsf{stage}_{\mathsf{id}} = 1$ :/This id finished bootstrap
        $\mathsf{id} \leftarrow \mathsf{id} + 1$
        $o \leftarrow \mathsf{LAC.Attest}^{\mathcal{M}}(\mathsf{prms}, \mathsf{hdl}, (p, (\mathsf{id}, \epsilon)), \epsilon)$
        $\mathsf{inreq} \leftarrow \mathsf{F}$; $\mathsf{m}' \leftarrow o$
        Return $(\mathsf{id}, \mathsf{inreq}, \mathsf{m}', \mathsf{st}_R)$
      Else:
        $\mathsf{inreq} \leftarrow \mathsf{F}$; $\mathsf{m}' \leftarrow o$
        Return $(\mathsf{id}, \mathsf{inreq}, \mathsf{m}', \mathsf{st}_R)$

/Execution (non-attested labels)
If $\mathsf{st}_R.\mathsf{stage} = 1$ :
    $(\mathsf{stage}_{\mathsf{id}}, \mathsf{id}, i) \leftarrow \mathsf{m}$
    If $i = \epsilon$ :/Move to next input (empty incoming message)
      $(\mathsf{id}_1, \ldots, \mathsf{id}_k) \leftarrow \mathsf{st}_R.\mathsf{IdList}$
      $\mathsf{inreq} \leftarrow \mathsf{T}$; $\mathsf{m}' \leftarrow \epsilon$
      $\mathsf{id} = \mathsf{id}_1$; $\mathsf{st}_R.\mathsf{IdList} \leftarrow (\mathsf{id}_2, \ldots, \mathsf{id}_k)$
      Return $(\mathsf{id}, \mathsf{inreq}, \mathsf{m}', \mathsf{st}_R)$
    Else:/Process input and send output
      $o \leftarrow \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, (q, \mathsf{id}), i)$
      $\mathsf{inreq} \leftarrow \mathsf{F}$; $\mathsf{m}' \leftarrow o$
      If $\mathsf{IdList} = [\,]$ : Then $\mathsf{st}_R.\mathsf{stage} \leftarrow 2$/No additional inputs
      Return $(\mathsf{id}, \mathsf{inreq}, \mathsf{m}', \mathsf{st}_R)$
```

Fig. 15. SMPC protocol untrusted scheduler.

of the AttKE. Due to the similarities of both proofs, we do not rewrite the whole proof, instead we highlight the differences coming from our different notion of attestation and our more general composition pattern.

Our proof is a sequence of four games $\mathsf{G0}'_{\mathsf{AttKE},\mathcal{A}}$ to $\mathsf{G3}'_{\mathsf{AttKE},\mathcal{A}}$. We highlight here how the sequence of games is constructed in relation with the original utility proof from [3].

FIRST GAME HOP. The first game $\mathsf{G0}'_{\mathsf{AttKE},\mathcal{A}}$ is the utility game presented in Figure 13. In [3] the first hop consists in adding a forgeAC event in the Send oracle to ensure that the initial segment of the trace witnessed by a local party matches the initial segment of a valid execution of the distant protocol. Here, similarly we add a forgeLAC event, making sure that the subtrace corresponding to the appropriate label matches a remote execution, the Send oracle is replaced by the one described in Figure 16.

The correctness of this game hop follows from the same arguments as the proof of Theorem 3 in [4].

```
Oracle Send($o^*, i$):
$o \leftarrow \mathsf{LAC.Verify}[\mathsf{st}_V^i](\mathsf{prms}, l^i, \mathsf{in}_{\mathsf{last}}^i, o^*)$
If $o = \perp$: Return $\perp$
If $b = 0$ then $\mathcal{M} \leftarrow \mathcal{M}$ else $\mathcal{M} \leftarrow \mathcal{M}'$
If $o \neq \perp \ \wedge\ \nexists \mathsf{hdl}$ s.t. $\mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}) = R_i^*$ .
 $\mathsf{Rev}(o : T_L^i) \sqsubseteq \mathsf{filter}[l^i]\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl})$: $\mathsf{forgeLAC} \leftarrow \mathsf{T}$
$\mathsf{m}^* \leftarrow\$ \mathsf{Loc}_{\mathsf{KE}}[\mathsf{st}_{\mathsf{KE}}^i](o)$
$\mathsf{in}_{\mathsf{last}}^i \leftarrow \mathsf{m}^*; T_L^i \leftarrow \mathsf{m}^* : o : T_L^i$
If $\mathsf{st}_{\mathsf{KE}}^i.\delta \in \{\mathsf{derived}, \mathsf{accept}\} \ \wedge\ \mathsf{st}_{\mathsf{KE}}^i.\mathsf{key} \notin \mathsf{fake}$:
    $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return $\mathsf{m}^*$
```

Fig. 16. Send oracle from $\mathsf{G1}'_{\mathsf{AttKE},\mathcal{A}}$

SECOND GAME HOP. The second game hop in [4] consist in replacing the remote machine by the simulator, provided by the minimum leakage property. The minimal leakage property is exactly the same in AC and LAC, and this second game hop is exactly the same.

This second game allows us to reason on the semantics of the original code instead of the compiled code executed in an IEE. This lets us take advantage of the semantics of parallel and sequential composition in the next game hop.

THIRD GAME HOP. In the third game hop, the crucial point is emulating a run of the protocol using the oracles from the AttKE security game. As in the proof of Theorem 3 from [4], we keep a list of instances of key exchanges related to the various programs, updated in the Load oracle. The NewSession oracle creates a new instance of $\mathsf{Rem}_{\mathsf{KE}}$ using the NewLoc oracle and the Send oracle uses the SendLoc oracle, exactly as in the original utility proof. The crucial modifications appear in the Run oracle and are presented in Figure 17.

We remark that for this last game hop to be valid, we crucially need both the fact that only the key and relevant parts of the key exchange state are passed through ϕ, which is ensured by the fact that the mapping function in the sequential composition is $(\phi_{\mathsf{key}}|\phi_1|\ldots|\phi_n)$. Additionally, the state of the key exchange has to be completely independent from the state of the other programs composed in parallel in order for us to be able to emulate it using the SendRem oracle. This property is ensured by the semantics of the parallel composition. With these remarks, we observe that the semantics of this third game is exactly the same as the semantics of $\mathsf{G2}'_{\mathsf{AttKE},\mathcal{A}}$, in a similar way to the utility proof in [4]. Finally, we observe that the adversary wins $\mathsf{G3}_{\mathsf{AttKE},\mathcal{A}}$ if it wins the AttKE security game (modulo the reduction simulating all non-AttKE oracles), which concludes the proof.

I Proof of Theorem 2

Proof. This proof is made by simulation. First, we present the construction of simulator $\mathcal{S}$, with the task of interacting with $\mathcal{A}$ on behalf of honest participants of the protocol, i.e., $\mathcal{M}$.Load, $\mathcal{M}$.Run and Send for parties 1 to k. We then present arguments for why

```
Oracle Run(hdl, (l, in)):
$(R_i^*, \mathsf{st}, j, \mathsf{stage}) \leftarrow \mathsf{List}[\mathsf{hdl}]$
If $(R_i^*, l_0) \in \mathsf{PrgList}$ and $R_i^* = \mathsf{LAC.Compile}(\langle\langle P|P_1 \dots |P_l\rangle_{(l_0,l_1,\dots,l_n)}; Q\rangle_{(\phi_{\mathsf{key}}|\phi_1|\dots|\phi_n),p,q})$:
    If $\mathsf{stage} = 1$:
        If $l = (p, l_0, \epsilon)$:
            If $\mathsf{st.finished}.l$: Return $(\mathsf{F}, \epsilon)$
            $o \leftarrow\$ \mathsf{SendRem}(\mathsf{in}, i, j)$
            Parse $(o, \mathsf{sid}, \delta, \mathsf{pid}) \leftarrow o$:
            If $\delta = \mathsf{accept}$:
                $\mathsf{st.finished}.l \leftarrow \mathsf{T}$
                $\mathsf{st}.l.\mathsf{key} \leftarrow \mathsf{TestRem}(i, j)$
        Else If $l = (p, l_k, l')$
            $o \leftarrow P_k[\mathsf{st}.l_k](l', \mathsf{in})$
            $\mathsf{st.finished}.l_k \leftarrow o.\mathsf{finished}$
            If $\mathsf{st.finished}.l_k$: $\mathsf{st}.l_k \leftarrow \phi_k(\mathsf{st}.l_k)$
        $o \leftarrow (\wedge_{i=0}^n \mathsf{st.finished}.l_i, o)$
        $(o^*, \mathsf{st}_{\mathcal{S}}) \leftarrow\$ \mathcal{S}_2(\mathsf{hdl}, R^*, l, \mathsf{in}, o, \mathsf{st}_{\mathcal{S}})$
        If $(\wedge_{i=0}^n \mathsf{st.finished}.l_i, o)$: $\mathsf{stage} = 2$
        $T_R^{\mathsf{hdl}} \leftarrow o^* : \mathsf{in} : T_R^{\mathsf{hdl}}$
    Else:
        $o \leftarrow\$ Q[\mathsf{st}](l, \mathsf{in})$
        $(o^*, \mathsf{st}_{\mathcal{S}}) \leftarrow\$ \mathcal{S}_2(\mathsf{hdl}, P, \phi, Q, R^*, \mathsf{in}, o, \mathsf{st}_{\mathcal{S}})$
Else:
    $(o, \mathsf{st}, \mathsf{st}_{\mathcal{S}}) \leftarrow\$ \mathcal{S}_3(\mathsf{hdl}, R^*, \mathsf{in}, \mathsf{st}, \mathsf{st}_{\mathcal{S}})$
$\mathsf{List}[\mathsf{hdl}] \leftarrow (R^*, \mathsf{st}, j, \mathsf{stage})$
Return $o^*$
```

Fig. 17. Run oracle from $\mathsf{G3}'_{\mathsf{AttKE},\mathcal{A}}$

adversary $\mathcal{A}$ cannot distinguish between this displayed interaction and the real world protocol execution.

Observe that, according to the experiment in Figure 2, despite being used in different contexts (e.g. the same $\mathcal{S}$ for emulating the machine and the presentation of outputs), the simulator can always distinguish to which call it is responding to. This is because it receives different inputs in different occasions, with exception of honest party initialization and output retrieval, whose orders are predictable (GetOutput will always provide $\mathcal{S}$ with an already initialized id). As such, for clarity of presentation, we describe our simulator in Figure 18 (local participants) and Figure 19 (remote machine) with different behavior for different calls. Notice that in this scenario there is no $\mathcal{M}$, however the simulator perfectly follows the description of $\mathcal{M}$ to emulate its behaviour. Following its description in Section 2, let $\mathsf{SMInit}(1^\lambda)$ be the initialization function of the security module, producing public parameters prms and internal state sk, and let $P^*[\mathsf{hdl}_{\mathsf{st}}, \mathsf{sk}](l, i)$ be the execution of compiled P^* given the internal state $\mathsf{hdl}_{\mathsf{st}}$ and private parameters sk, according to the description of the security module, producing (possibly attested) output o^*.

The behaviour detailed in $\mathcal{S}$ does not trivially entail indistinguishability from the real world on all cases. The two main differences between how the simulator handles calls and how the same instructions would be executed in the real world are highlighted in the presented figures, and are now further detailed.

– The simulator is replacing the exchanged keys associated with honest participants with randomly generated ones (fake), and using them throughout the second stage of the protocol.

```
$\underline{\mathcal{S}(1^\lambda)}$:/parameter initialization
$\mathsf{prms}, \mathsf{st.sk} \leftarrow\$ \mathsf{SMInit}(1^\lambda)$
$\mathsf{st}.\lambda \leftarrow 1^\lambda$; $\mathsf{st.hdl} \leftarrow 0$; $\mathsf{st.fake} \leftarrow [\,]$
Return $(\mathsf{st}, \mathsf{st.prms})$

$\underline{\mathcal{S}(\mathsf{st}, \mathsf{id})}$:/party setup
$(\mathsf{st}_L, \mathsf{Rem}_{\mathsf{KE}}) \leftarrow \mathsf{Setup}_{\mathsf{KE}}(\mathsf{st}.\lambda, \mathsf{id})$
$\mathsf{st.id.st}_L \leftarrow \mathsf{st}_L$
$\mathsf{st.stage} \leftarrow 0$
Return $(\mathsf{st}, \mathsf{Rem}_{\mathsf{KE}})$

$\underline{\mathcal{S}(\mathsf{st}, \mathsf{id})}$:/party initialization
$\mathsf{st.id.InList} \leftarrow [\,]$; $\mathsf{st.id.stage} \leftarrow 0$;
$\mathsf{st.id.seq_{in}} \leftarrow 0$; $\mathsf{st.id.seq_{out}} \leftarrow 1$; $\mathsf{st.id.in_{last}} \leftarrow \epsilon$
$(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
$\mathsf{st}.P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n}\, ; \,\mathsf{Box}\langle \mathcal{F}, \Lambda \rangle \,\rangle_{\phi_{\mathsf{key}}, p, q}$
$\mathsf{st}.L \leftarrow \{(p, (\mathsf{id}, \epsilon)), (q, \mathsf{id})\}$
$\mathsf{st.id.st}_V \leftarrow (\mathsf{st}.P, \mathsf{st}.L)$
$L^* \leftarrow \{(p, (1, \epsilon)), \ldots, \{(p, (n, \epsilon))\}$
$\mathsf{st}.P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, st.P, L^*)$
$\mathsf{st.id.InLeak} \leftarrow [\,]$; $\mathsf{st.id.InList} \leftarrow [\,]$
Return st

$\underline{\mathcal{S}(\mathsf{st}, l, \mathsf{id})}$:/add inputs
$\mathsf{st_{id}.InLeak[id]} \leftarrow \mathsf{st_{id}.InLeak[id]} + [l]$
Return st

$\underline{\mathcal{S}(\mathsf{st}, \mathsf{id})}$:/output retrieval
Return $(\mathsf{st.id.seq_{out}}/2) + 1$
```

```
$\underline{\mathcal{S}^{\mathsf{Fun}}(\mathsf{st}, \mathsf{id}, \mathsf{m})}$:/emulate local participant id
If $\mathsf{st.id.stage} = 0$ :
  $(i, \mathsf{st.id.st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st.prms}, (p, (\mathsf{id}, \epsilon)), \mathsf{st.id.in_{last}}, \mathsf{m}, \mathsf{st.id.st}_V)$
  If $i = \perp$: Return $\perp$
  $(o, \mathsf{st.id.st}_L) \leftarrow\$ \mathsf{Loc_{KE}}(\mathsf{st.id.st}_L, i)$
  $\mathsf{st.id.in_{last}} \leftarrow o$
  If $\mathsf{st.id.st}_L.\mathsf{key} \notin \mathsf{st.fake} \wedge \mathsf{st.id.st}_L.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
      $\mathsf{key}^* \leftarrow\$ \{0,1\}^{\mathsf{st}.\lambda}$
      $\mathsf{st.fake} \leftarrow (\mathsf{st.id.st}_L.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
  If $\mathsf{st.id.st}_L.\delta = \mathsf{accept}$: stage $\leftarrow 1$
  $\mathsf{m}' \leftarrow (\mathsf{st.id.stage}, \mathsf{id}, o)$
  Return $(\mathsf{st}, \mathsf{m}')$
If $\mathsf{st.id.stage} = 1$ :
  If $\mathsf{m} = \epsilon$ :
    $l \leftarrow \mathsf{st.id.InLeak}[0]$
    $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st.id.InLeak}$
    $\mathsf{st.id.InLeak} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_{k\text{-}1})$
    $\mathsf{in} \leftarrow \{0\}^l$
    $o \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st.id.st}_L.\mathsf{key}), (\mathsf{st.id.seq_{in}}, \mathsf{in}))$
    $\mathsf{st.id.InList}[\mathsf{st.id.seq_{in}}] \leftarrow o$
    $\mathsf{st.id.in_{last}} \leftarrow o$
    $\mathsf{st.id.seq_{in}} \leftarrow \mathsf{st.id.seq_{in}} + 2$
    $\mathsf{m}' \leftarrow (\mathsf{st.id.stage}, \mathsf{id}, o)$
    Return $(\mathsf{st}, \mathsf{m}')$
  Else:
    $\mathsf{m}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st.id.st}_L.\mathsf{key}), \mathsf{m})$
    If $\mathsf{m}' = (\mathsf{st.id.seq_{out}}, \mathsf{out}')$ :
      $\mathsf{st.id.seq_{out}} \leftarrow \mathsf{st.id.seq_{out}} + 2$
      $\mathsf{m}' \leftarrow (\mathsf{st.id.stage}, \mathsf{id}, \epsilon)$
      Return $(\mathsf{st}, \mathsf{m}')$
    Else: Return $\perp$
```

Fig. 18. Description of simulator $\mathcal{S}$ with respect to emulating local participants.

– Instead of the honest participant's inputs and outputs, the simulator is encrypting strings of 0s with the same length as the real-world values (obtained by Lin and Lout).

We now argue that, nevertheless, this provides an indistinguishable view for any $\mathcal{A}$. We prove this in three game hops from the real world, from Figure 20 to Figure 23. The first hop will replace $\mathcal{M}$ with the slightly different $\mathcal{M}'$, which replaces keys exchanged by honest participants by freshly generated keys (in exactly the same way the simulator is doing it). The correctness of this hop follows from the utility theorem, using a hybrid argument to replace keys of all k honest parties. Afterwards, we replace the encrypted inputs/outputs of honest parties, by encrypting dummy payloads of the correct length. The correctness of this hop follows from the indistinguishability of the underlying authenticated encryption scheme. Finally, we restrict the possibility of $\mathcal{A}$ to produce a forged encryption, by accordingly establishing a *bad* event. The correctness of this final hop follows from the unforgeability of the underlying authenticated encryption scheme.

The first game (Figure 20) is simply the real game expanded with the protocol instantiation. In this setting, whenever the adversary sets k as 0, i.e. corrupts all participants, the simulator already produces an indistinguishable view. In this case there are no honest inputs/outputs, so the simulator has access to all information and can therefore execute the protocol without replacing any keys and without encrypting any dummy

```
$\mathcal{S}(\mathsf{st}, \mathsf{Pub}, P)$:/$\mathcal{M}$.Load
$\mathsf{st.hdl} \leftarrow \mathsf{st.hdl} + 1$
For $i \in L$: $\mathsf{seq}[i] \leftarrow 0$
$\mathsf{st.HdlList} \leftarrow (\mathsf{st.hdl}, \mathsf{seq}, \epsilon)$
Return $\mathsf{st.hdl}$

$\mathcal{S}^{\mathsf{Fun}}(\mathsf{st}, \mathsf{hdl}, \mathsf{m})$:/$\mathcal{M}$.Run
$(P^*, \mathsf{seq}, \mathsf{st_{hdl}}) \leftarrow \mathsf{st.HdlList}[\mathsf{hdl}]$
If $P^* = \mathsf{st}.P^*$:/The agreed protocol.
    If $(p, (\mathsf{id}, \epsilon)) \notin \mathsf{st}.L$: Return $\perp$
    If $\mathsf{st_{hdl}}[\mathsf{id}].\mathsf{stage} = 0$ :
        $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{m}$
        $\mathsf{m}' \leftarrow\$ P^*[\mathsf{st_{hdl}}, \mathsf{st.sk}](\mathsf{id}, \mathsf{m})$
        If $\mathsf{st.id} \neq \epsilon \wedge \mathsf{st_{hdl}}[\mathsf{id}].\mathsf{key} \notin \mathsf{st.fake} \wedge \mathsf{st_{hdl}}[\mathsf{id}].\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
            $\mathsf{key}^* \leftarrow\$ \{0,1\}^{\mathsf{st}.\lambda}$
            $\mathsf{st.fake} \leftarrow (\mathsf{st_{hdl}}[\mathsf{id}].\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
    Else If $\mathsf{st_{hdl}}[\mathsf{id}].\mathsf{stage} = 1$ :
        $(\mathsf{seq_{in}}, \mathsf{id}, \mathsf{in}) \leftarrow \mathsf{m}$
        If $(\mathsf{seq}[\mathsf{id}] \neq \mathsf{seq_{in}})$: Return $\perp$
        If $\mathsf{st.id} \neq \epsilon$:/Honest participant
            If $\mathsf{st.id.InList}[\mathsf{seq}[\mathsf{id}]] \neq \mathsf{in}$: Return $\perp$
            $l \leftarrow \mathsf{Fun}(\mathsf{honest}, \mathsf{id}, \epsilon)$
            $\mathsf{out} \leftarrow \{0\}^l$
            $\mathsf{m}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st_{hdl}}.\mathsf{key}[\mathsf{id}]), (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{in}))$
        Else:/Corrupt participant
            $\mathsf{in}^* \leftarrow\$ \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st_{hdl}}.\mathsf{key}[\mathsf{id}]), (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{in}))$
            $\mathsf{out} \leftarrow \mathsf{Fun}(\mathsf{corrupt}, \mathsf{id}, \mathsf{in}^*)$
            $\mathsf{m}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st_{hdl}}.\mathsf{key}[\mathsf{id}]), (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{out}))$
        $\mathsf{seq}[\mathsf{id}] \leftarrow \mathsf{seq}[\mathsf{id}] + 2$
Else:/Any other program on $\mathcal{M}$.
    $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{m}$
     $\mathsf{m}' \leftarrow\$ P^*[\mathsf{st_{hdl}}, \mathsf{st.sk}](\mathsf{id}, \mathsf{m})$
$\mathsf{st.HdlList}[\mathsf{hdl}] \leftarrow (P, \mathsf{seq}, \mathsf{st_{hdl}})$
Return $(\mathsf{st}, \mathsf{m}')$
```

Fig. 19. Description of simulator $\mathcal{S}$ with respect to emulating the remote machine.

payloads ($\mathsf{st.id} = \epsilon$ for all id), executing Fun whenever a corrupt input is provided to produce the corresponding output. As such, the following steps will only refer to situations in which $k \neq 0$, where indistinguishability is not yet established.

In the second game $\mathbf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)$ (Figure 21), we replace the machine in the ideal world with the machine M' of the Utility game for which $b = 1$. This machine performs exactly what the simulator is doing with the list fake, i.e., replacing keys for the first k participants whenever they finish the first stage of the protocol (the key exchange). This is possible via two steps.

Fix identity $\mathsf{id} = 1$. We can replace the behaviour of $\mathcal{M}$ in $\mathbf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda)$ regarding this participant using the Utility Theorem 3, for which $l^* = \mathsf{id}$. In this scenario, the key (both in $\mathcal{M}'$.Run and in Send) for that particular participant will be replaced by a fake one and stored in fake (as described in $\mathbf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)$, but only for $\mathsf{id} = 1$). The advantage gained by the adversary in this intermediate step is bound by its advantage in winning the experiment in Figure 13, by providing

$$\mathsf{Rem}^2_{\mathsf{KE}}, (p, (2, \epsilon)), (q, 2), \ldots \mathsf{Rem}^n_{\mathsf{KE}}, (p, (n, \epsilon)), (q, n), (p, (1, \epsilon)), \mathsf{Box}\langle \mathcal{F}, \Lambda\rangle, \phi_{\mathsf{key}}$$

To NewSession on every call.

Now observe that, for any scenario in which m participants have had their keys replaced by fake ones, it is possible to apply the same Utility theorem for replacing the

```
$\mathbf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(\mathsf{st}_\mathcal{A}, k) \leftarrow\$ \mathcal{A}(\mathsf{prms})$
For id $\in [1..k]$:
  $(\mathsf{st}_\mathsf{id}.\mathsf{st}_L, \mathsf{st}_\mathsf{id}.\mathsf{pub}) \leftarrow \mathsf{Setup}_\mathsf{KE}(1^\lambda, \mathsf{id})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_k)$
For id $\in [k+1..n]$:
  $(\mathsf{st}_\mathcal{A}, \mathsf{pub}_\mathsf{id}) \leftarrow\$ \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{id}, \mathsf{Pub})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_n)$
For id $\in [1..k]$:
  $\mathsf{st}_\mathsf{id}.\mathsf{InList} \leftarrow [\,]$; $\mathsf{st}_\mathsf{id}.\mathsf{stage} \leftarrow 0$;
  $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} \leftarrow 0$; $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} \leftarrow 1$; $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow \epsilon$
  $(\mathsf{Rem}^1_\mathsf{KE}, \ldots, \mathsf{Rem}^n_\mathsf{KE}) \leftarrow \mathsf{Pub}$
  $P \leftarrow \langle\, \langle \mathsf{Rem}^1_\mathsf{KE}, \ldots, \mathsf{Rem}^n_\mathsf{KE} \rangle_{1,\ldots,n}\, ;\ \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle\, \rangle_{\phi_\mathsf{key}, p, q}$
  $L \leftarrow \{(p, (\mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)), (q, \mathsf{st}_\mathsf{id}.\mathsf{id})\}$
  $\mathsf{st}_\mathsf{id}.\mathsf{st}_V \leftarrow (P, L)$
$b \leftarrow\$ \mathcal{A}^\mathcal{O}(\mathsf{st}_\mathcal{A})$

Oracle $\mathsf{SetInput}(\mathsf{in}, \mathsf{id})$:
If id $\notin [1..k]$ Return $\perp$
$\mathsf{st}_\mathsf{id}.\mathsf{InList} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{InList} + [\mathsf{in}]$

Oracle $\mathsf{Load}(P)$:
Return $\mathcal{M}.\mathsf{Load}(P)$

Oracle $\mathsf{Run}(\mathsf{hdl}, l, \mathsf{m})$:
Return $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, l, \mathsf{m})$

Oracle $\mathsf{Send}(\mathsf{id}, \mathsf{m})$:
If id $\notin [1..k]$ Return $\perp$
If $\mathsf{st}_\mathsf{id}.\mathsf{stage} = 0$ :
  $(i, \mathsf{st}_\mathsf{id}.\mathsf{st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st}_\mathsf{id}.\mathsf{prms}, (p, (\mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)), \mathsf{in}_\mathsf{last}, \mathsf{m}, \mathsf{st}_V)$
  If $i = \perp$: Return $\perp$
  $(o, \mathsf{st}_\mathsf{id}.\mathsf{st}_L) \leftarrow\$ \mathsf{Loc}_\mathsf{KE}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L, i)$
  $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow o$
  If $(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{st}_\mathsf{KE}.\delta) = \mathsf{accept}$ : Then stage $\leftarrow 1$
  $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, o)$
  Return $\mathsf{m}'$
If $\mathsf{st}_\mathsf{id}.\mathsf{stage} = 1$ :
  If $\mathsf{m} = \epsilon$ :
    $\mathsf{in} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{InList}[0]$
    $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{ListIn}_\mathsf{id}$
    $\mathsf{st}_\mathsf{id}.\mathsf{ListIn}_\mathsf{id} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_{k\text{-}1})$
    $o \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}, (\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in}, \mathsf{in}))$
    $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow o$
    $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} + 2$
    $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, o)$
    Return $\mathsf{m}'$
  Else:
    $\mathsf{m}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}, \mathsf{m})$
    If $\mathsf{m}' = (\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out}, \mathsf{out}')$ :
      $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} + 2$
      $\mathsf{st}_\mathsf{id}.\mathsf{out} \leftarrow \mathsf{out}'$
      $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)$
      Return $\mathsf{m}'$
Else: Return $\perp$

Oracle $\mathsf{GetOutput}(\mathsf{id})$:
If id $\notin [1..k]$ Return $\perp$
Return $\mathsf{st}_\mathsf{id}.\mathsf{out}$
```

Fig. 20. Real world expanded.

keys of $m+1$ participants. In order to replace all k keys, we are therefore required to apply the same Utility theorem k times, and thus

$$\Pr[\,\mathsf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \mathsf{Adv}^{\mathsf{UT}}_{\mathsf{AttKE},\mathcal{A}}(\lambda) * \mathsf{k}.$$

In the third game $\mathbf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$ (Figure 22), we open the machine M' and change its behaviour for instances of running program P^* on the second stage as follows:

– Upon receiving an honest participant input, instead of using it for computing Fun instead uses the first element of ListIn.
– When producing an honest participant output, instead of returning an encryption of the value received from F, it stores the value from F on an output list for this identity OutList and returns an encryption of zeros of the same length as the output.

Similarly, on the local side for instances running the second stage:

– When called for presenting the input, instead of encrypting the actual input, it stores it on a list of inputs InList and encrypts a string of zeros of the same length.
– Upon receiving an output, instead of decrypting and storing it on ListOut, it retrieves the value of OutList and stores it on ListOut.

```
$\mathbf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$\mathsf{prms} \leftarrow\!\!\$\ \mathcal{M}'.\mathsf{Init}(1^\lambda)$
$\mathsf{fake} \leftarrow [\,]$
$(\mathsf{st}_\mathcal{A}, k) \leftarrow\!\!\$\ \mathcal{A}(\mathsf{prms})$
For id $\in [1..k]$:
    $(\mathsf{st}_\mathsf{id}.\mathsf{st}_L, \mathsf{st}_\mathsf{id}.\mathsf{pub}) \leftarrow \mathsf{Setup}_\mathsf{KE}(1^\lambda, \mathsf{id})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_k)$
For id $\in [k+1..n]$:
    $(\mathsf{st}_\mathcal{A}, \mathsf{pub}_\mathsf{id}) \leftarrow\!\!\$\ \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{id}, \mathsf{Pub})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_n)$
For id $\in [1..k]$:
    $\mathsf{st}_\mathsf{id}.\mathsf{InList} \leftarrow [\,]$; $\mathsf{st}_\mathsf{id}.\mathsf{stage} \leftarrow 0$;
    $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} \leftarrow 0$; $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} \leftarrow 1$; $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow \epsilon$
    $(\mathsf{Rem}^1_\mathsf{KE}, \ldots, \mathsf{Rem}^n_\mathsf{KE}) \leftarrow \mathsf{Pub}$
    $P \leftarrow \langle\, \langle \mathsf{Rem}^1_\mathsf{KE}, \ldots, \mathsf{Rem}^n_\mathsf{KE} \rangle_{1,\ldots,n}\ ;\ \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle\, \rangle_{\phi_\mathsf{key}, p, q}$
    $L \leftarrow \{(p, (\mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)), (q, \mathsf{st}_\mathsf{id}.\mathsf{id})\}$
    $\mathsf{st}_\mathsf{id}.\mathsf{st}_V \leftarrow (P, L)$
    $P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, P, L)$
$b \leftarrow\!\!\$\ \mathcal{A}^\mathcal{O}(\mathsf{st}_\mathcal{A})$

Oracle $\mathsf{SetInput}(\mathsf{in}, \mathsf{id})$:
If id $\notin [1..k]$ Return $\perp$
$\mathsf{st}_\mathsf{id}.\mathsf{InList} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{InList} + [\mathsf{in}]$

Oracle $\mathsf{Load}(P)$:
Return $\mathcal{M}'.\mathsf{Load}(P)$

Oracle $\mathsf{Run}(\mathsf{hdl}, l, \mathsf{m})$:
$\mathsf{flag} \leftarrow \mathsf{F}$
If $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = P^*$: $\mathsf{flag} \leftarrow \mathsf{T}$
Return $\mathcal{M}'.\mathsf{Run}(\mathsf{hdl}, l, \mathsf{m}, \mathsf{flag}, \mathsf{fake})$
```

```
Oracle $\mathsf{Send}(\mathsf{id}, \mathsf{m})$:
If id $\notin [1..k]$ Return $\perp$
If $\mathsf{st}_\mathsf{id}.\mathsf{stage} = 0$ :
    $(i, \mathsf{st}_\mathsf{id}.\mathsf{st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st}_\mathsf{id}.\mathsf{prms}, (p, (\mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)), \mathsf{in}_\mathsf{last}, \mathsf{m}, \mathsf{st}_V)$
    If $i = \perp$: Return $\perp$
    $(o, \mathsf{st}_\mathsf{id}.\mathsf{st}_L) \leftarrow\!\!\$\ \mathsf{Loc}_\mathsf{KE}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L, i)$
    $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow o$
    If $\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}_\mathsf{id}.\mathsf{st}_L.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
        $\mathsf{key}^* \leftarrow\!\!\$\ \{0,1\}^\lambda$
        $\mathsf{st.fake} \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
    If $(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{st}_\mathsf{KE}.\delta) = \mathsf{accept}$ : Then stage $\leftarrow 1$
    $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, o)$
    Return $\mathsf{m}'$
If $\mathsf{st}_\mathsf{id}.\mathsf{stage} = 1$ :
    If $\mathsf{m} = \epsilon$ :
        $\mathsf{in} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{InList}[0]$
        $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{ListIn}_\mathsf{id}$
        $\mathsf{st}_\mathsf{id}.\mathsf{ListIn}_\mathsf{id} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_{k\text{-}1})$
        $o \leftarrow\!\!\$\ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}), (\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in}, \mathsf{in}))$
        $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow o$
        $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} + 2$
        $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, o)$
        Return $\mathsf{m}'$
    Else:
        $\mathsf{m}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}), \mathsf{m})$
        If $\mathsf{m}' = (\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out}, \mathsf{out}')$ :
            $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} + 2$
            $\mathsf{st}_\mathsf{id}.\mathsf{out} \leftarrow \mathsf{out}'$
            $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)$
            Return $\mathsf{m}'$
Else: Return $\perp$

Oracle $\mathsf{GetOutput}(\mathsf{id})$:
If id $\notin [1..k]$ Return $\perp$
Return $\mathsf{st}_\mathsf{id}.\mathsf{out}$
```

Fig. 21. First hop of the proof.

We upper bound the distance between these two games, by constructing an adversary $\mathcal{B}$ against the indistinguishability of encryption scheme Λ such that

$$\Pr[\,\mathsf{G1}_{\mathcal{F},\pi,\mathcal{M}'}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G2}_{\mathcal{F},\pi}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \mathsf{Adv}^{\mathsf{IND}}_{\Lambda,\mathcal{B}}(\lambda) * k * 2I$$

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$ as follows: it first has to try and guess which message will be used to distinguish. Let I be the maximum number of inputs adversary $\mathcal{A}$ chooses to input for any participant. It samples uniformly from $[1..k]$ a participant p, and from $[1..(I*2)]$ a message m. Since every input produces an output, we establish that

- If $m \in [1..I]$, $\mathcal{B}$ picked the m-th input.
- If $m \in [I+1, \ldots, (I*2)]$, $\mathcal{B}$ picked the $\frac{m}{I}$-th output.

and proceed accordingly. $\mathcal{B}$ replaces all calls for encryption/decryption for inputs/outputs of participant p with similar calls to $\Lambda.\mathsf{Enc}$ and $\Lambda.\mathsf{Dec}$, with exception of the following. If $m \in [1..I]$, whenever $\mathsf{Send}(\mathsf{id}, \mathsf{m})$ for $\mathsf{id} = p$ is called for the m-th time on the second stage, $\mathcal{B}$ challenges $\mathsf{IND}^{\Lambda,\mathcal{B}}(1^\lambda)$ with message

$$((\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \mathsf{in}), (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \{0\}^{|\mathsf{in}|}))$$

Otherwise, whenever $\mathsf{Run}(\mathsf{hdl}, l, \mathsf{m})$ for $l = p$ and $P = P^*$ (the agreed protocol) is called for the $\frac{m}{I}$-th time on the second stage, $\mathcal{B}$ challenges $\mathsf{IND}^{\Lambda,\mathcal{B}}(1^\lambda)$ with message

$$((\mathsf{seq}[\mathsf{id}] + 1, \mathsf{out}), (\mathsf{seq}[\mathsf{id}] + 1, \{0\}^{|\mathsf{out}|}))$$

Observe that any advantage $\mathcal{B}$ acquires in this transformation can be effectively used to distinguish between $\mathsf{G1}_{\mathcal{F},\pi,\mathcal{M}'}(1^\lambda)$ and $\mathsf{G2}_{\mathcal{F},\pi}(1^\lambda)$, since the only difference between the two games is the encryption of either the first message (the real value) or the second (the dummy payload with the same length). The two games are identical modulo this difference.

In the fourth game $\mathbf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$ (Figure 23), the adversary loses whenever $\mathsf{authForge}$ event occurs. Intuitively, this event corresponds to the adversary producing an encryption that was not produced by either Send or $\mathcal{M}.\mathsf{Run}$ for $\mathsf{id} \in [1..k]$ (honest participant), and hence constitutes a forgery with respect to Λ. Given that the two games are identical until this event occurs, we have that

$$\Pr[\,\mathsf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{authForge}\,]\,.$$

We upper bound the distance between these two games, by constructing an adversary $\mathcal{C}$ against the existential unforgeability of encryption scheme Λ such that

$$\Pr[\,\mathsf{authForge}\,] \leq \mathsf{Adv}^{\mathsf{UF}}_{\Lambda,\mathcal{C}}(\lambda) * k$$

Adversary $\mathcal{C}$ simulates the environment of $\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$ as follows: it first has to try and guess which session will produce the forgery. As such, it samples uniformly from $[1..k]$ a participant p and replaces the key generated for honest participant p (before adding to fake) with the key generated by $\Lambda.\mathsf{Gen}$. From there on, every time an encryption/decryption is requested for p, the same operation will be requested to $\Lambda.\mathsf{Enc}$ and $\Lambda.\mathsf{Dec}$, respectively.

When $\mathsf{authForge}$ is set, according to the rules of $\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$, algorithm $\mathcal{C}$ outputs candidate encryption m. It remains to show that this is a valid forgery. To see this, first observe that this is indeed a valid encryption, as decryption is performed on this value immediately before $\mathsf{authForge}$ occurs. It suffices to establish that message m could not have been queried from the Λ oracle. Access to this oracle is only permitted on the encryption of inputs for this participant, and on outputs to this participant (when executing Run). From the construction of these operations and the sequence numbers they entail, we know that producing such an encryption would only occur via the inclusion of m in $\mathsf{authList}$. Since we know this is not the case, m could not have been queried to the encryption oracle. We conclude therefore that $\mathcal{C}$ outputs a valid forgery whenever $\mathsf{authForge}$ occurs.

Finally, we argue that the behavior displayed by the simulator is indistinguishable to what adversary $\mathcal{A}$ observes in game $\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$. This is the case because the simulator no longer has private information to which he has no access to. In $\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$, only the length of honest inputs and outputs is required to emulate their private inputs and outputs, to which $\mathcal{S}$ has access to via Lin and Lout. Additionally, the simulator uses the same message sequence numbers to prevent $\mathcal{A}$ from forcing an execution that deviates from the order in which inputs are provided and outputs are retrieved. Since the executions are the same for all other aspects (including key replacements to fake and exclusion of forged encryptions), $\mathcal{A}$ is provided the same view in both worlds.

Let

$$\mathsf{Adv}^{\mathsf{Distinguish}}_{\mathcal{F},\mathcal{A}} = \Pr[\,\mathsf{Real}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{Ideal}_{\mathcal{F},\pi,\mathcal{A},\mathcal{S}}(1^\lambda) \Rightarrow \mathsf{T}\,]$$

To conclude, we have that

$$\begin{aligned}\mathsf{Adv}^{\mathsf{Distinguish}}_{\mathcal{F},\mathcal{A}} &= \Pr[\,\mathsf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda)\,] - \Pr[\,\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)\,]\\ &= (\Pr[\,\mathsf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda)\,] - \Pr[\,\mathsf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)\,]) + (\Pr[\,\mathsf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)\,] - \\ &\Pr[\,\mathsf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)\,]) + (\Pr[\,\mathsf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)\,] - \Pr[\,\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)\,])\\ &\le \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{UT},\mathcal{A}}(\lambda) * k + \mathsf{Adv}^{\mathsf{IND}}_{\Lambda,\mathcal{B}}(\lambda) * k * 2I + \Pr[\mathsf{forgeAuth}]\\ &\le \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{UT},\mathcal{A}}(\lambda) * k + \mathsf{Adv}^{\mathsf{IND}}_{\Lambda,\mathcal{B}}(\lambda) * k * 2I + \mathsf{Adv}^{\mathsf{UF}}_{\Lambda,\mathcal{C}}(\lambda) * k\end{aligned}$$

and Theorem 2 follows.

J Side channels and software resilient against timing attacks

Recent works [42,17] have pointed out that IEE-enabled systems such as Intel's SGX do not offer more protection against side-channel attacks than traditional microprocessors. This is a relevant concern, since the IEE trust model which we also adopt in this paper admits that the code outside IEEs is potentially malicious and that the machine is under the control of an untrusted party. We believe that there are two aspects to this problem that should be considered separately. The first aspect is related to the production of the IEE-enabled hardware/firmware itself and the protection of the long-term secrets that are used by the attestation security module. If the computations performed by the attestation infrastructure itself are vulnerable to side-channel attacks, then there is nothing that can be done at the protocol design/implementation level. This aspect of trust is within the remit of the equipment manufacturers.

An orthogonal issue is the possibility that software running inside an IEE leaks part of its state or short-term secrets via side channels. Here one should distinguish between software observations and hardware/physical observations. In the former, software co-located in the machine observes timing channels based on memory access patterns, control flow, branch prediction, cache-based based attacks [17], page-fault side channels [42], etc. Protection against this type of side-channel attacks has been widely studied in the practical crypto community, where a consensus exists that writing so-called *constant-time* software is the most effective countermeasure [8,34]. As mentioned above, constant-time software has the property that the entire sequence of memory addresses (in both data and code memory) accessed by a program can be predicted in advance from public inputs, e.g., the length of messages. When it comes to hardware/physical side-channel attacks such as those relying on temperature measurements, power analysis, or electromagnetic radiation, the effectiveness of software countermeasures is very limited, and improving hardware defenses again implies obtaining additional guarantees from the equipment manufacturer.

Our implementation sgx-mpc-nacl enforces a strict constant-time policy that is consistent with the IEE trust model. As such, to provide a protocol that is fully constant-time, one must ensure that the executed functionality is also constant-time. Recent work in the area of formal verification area sheds new light how this can be achieved over low-level code in a fully automatic way [1].

```
**G2**$_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$(\mathsf{prms}, \mathsf{sk}) \leftarrow\$ \mathsf{SMInit}(1^\lambda)$
$\mathsf{hdl} \leftarrow 0$
$\mathsf{fake} \leftarrow [\,]$
$(\mathsf{st}_\mathcal{A}, k) \leftarrow\$ \mathcal{A}(\mathsf{prms})$
For id $\in [1..k]$:
    $(\mathsf{st}_\mathsf{id}.\mathsf{st}_L, \mathsf{st}_\mathsf{id}.\mathsf{pub}) \leftarrow \mathsf{Setup}_\mathsf{KE}(1^\lambda, \mathsf{id})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_k)$
For id $\in [k+1..n]$:
    $(\mathsf{st}_\mathcal{A}, \mathsf{pub}_\mathsf{id}) \leftarrow\$ \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{id}, \mathsf{Pub})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_n)$
For id $\in [1..k]$:
    $\mathsf{st}_\mathsf{id}.\mathsf{ListIn} \leftarrow [\,]$; $\mathsf{st}_\mathsf{id}.\mathsf{ListOut} \leftarrow [\,]$; $\mathsf{st}_\mathsf{id}.\mathsf{stage} \leftarrow 0$
    $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} \leftarrow 0$; $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} \leftarrow 1$; $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow \epsilon$
    $(\mathsf{Rem}^1_\mathsf{KE}, \ldots, \mathsf{Rem}^n_\mathsf{KE}) \leftarrow \mathsf{Pub}$
    $P \leftarrow \langle\, \langle \mathsf{Rem}^1_\mathsf{KE}, \ldots, \mathsf{Rem}^n_\mathsf{KE} \rangle_{1,\ldots,n}\ ;\ \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle\, \rangle_{\phi_\mathsf{key}, p, q}$
    $L \leftarrow \{(p, (\mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)), (q, \mathsf{st}_\mathsf{id}.\mathsf{id})\}$
    $\mathsf{st}_\mathsf{id}.\mathsf{st}_V \leftarrow (P, L)$
    $P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, P, L)$
$b \leftarrow\$ \mathcal{A}^\mathcal{O}(\mathsf{st}_\mathcal{A})$

**Oracle** Run$(\mathsf{hdl}, l, \mathsf{m})$:
$(P, \mathsf{seq}, \mathsf{st}_\mathsf{hdl}) \leftarrow \mathsf{HdlList}[\mathsf{hdl}]$
If $P = P^*$://The agreed protocol.
    If $(p, (\mathsf{id}, \epsilon)) \notin L$: Return $\perp$
    If $\mathsf{st}_\mathsf{hdl}[\mathsf{id}].\mathsf{stage} = 0$ :
        $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{m}$
        $\mathsf{m}' \leftarrow\$ P^*[\mathsf{st}_\mathsf{hdl}, \mathsf{sk}](\mathsf{id}, \mathsf{m})$
        If $\mathsf{st}_\mathsf{id} \neq \epsilon \wedge \mathsf{st}_\mathsf{hdl}[\mathsf{id}].\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}_\mathsf{hdl}[\mathsf{id}].\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
            $\mathsf{key}^* \leftarrow\$ \{0,1\}^{1^\lambda}$
            $\mathsf{st}.\mathsf{fake} \leftarrow (\mathsf{st}_\mathsf{hdl}[\mathsf{id}].\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
    Else If $\mathsf{st}_\mathsf{hdl}[\mathsf{id}].\mathsf{stage} = 1$ :
        $(\mathsf{seq}_\mathsf{in}, \mathsf{id}, \mathsf{in}) \leftarrow \mathsf{m}$
        If $(\mathsf{seq}[\mathsf{id}] \neq \mathsf{seq}_\mathsf{in})$: Return $\perp$
        If $\mathsf{st}_\mathsf{id} \neq \epsilon$://Honest participant
            $\mathsf{m}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}), \mathsf{in})$
            If $\mathsf{m}' = (\mathsf{seq}[\mathsf{id}], \mathsf{out}')$ :
                $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_\mathsf{F}](\mathsf{id}, \mathsf{InList}[\mathsf{seq}[\mathsf{id}]])$
                $\mathsf{st}_\mathsf{id}.\mathsf{OutList}[\mathsf{seq}[\mathsf{id}] + 1] \leftarrow \mathsf{out}$
                $\mathsf{m}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_\mathsf{hdl}.\mathsf{key}[\mathsf{id}]), (\mathsf{seq}[\mathsf{id}] + 1, \{0\}^{|\mathsf{out}|}))$
        Else://Corrupt participant
            $\mathsf{in}^* \leftarrow\$ \Lambda.\mathsf{Dec}(\mathsf{st}_\mathsf{hdl}.\mathsf{key}[\mathsf{id}], (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{in}))$
            $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_\mathsf{F}](\mathsf{id}, \mathsf{in})$
            $\mathsf{m}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{st}_\mathsf{hdl}.\mathsf{key}[\mathsf{id}], (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{out}))$
        $\mathsf{seq}[\mathsf{id}] \leftarrow \mathsf{seq}[\mathsf{id}] + 2$
Else://Any other program on $\mathcal{M}$.
    $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{m}$
    $\mathsf{m}' \leftarrow\$ P^*[\mathsf{st}_\mathsf{hdl}, \mathsf{st}.\mathsf{sk}](\mathsf{id}, \mathsf{m})$
$\mathsf{HdlList}[\mathsf{hdl}] \leftarrow (P, \mathsf{seq}, \mathsf{st}_\mathsf{hdl})$
Return $\mathsf{m}'$

**Oracle** Load$(P)$:
$\mathsf{hdl} \leftarrow \mathsf{hdl} + 1$
For $i \in L$: $\mathsf{seq}[i] \leftarrow 0$
$\mathsf{HdlList} \leftarrow (\mathsf{hdl}, \mathsf{seq}, \epsilon)$
Return hdl

**Oracle** SetInput$(\mathsf{in}, \mathsf{id})$:
If id $\notin [1..k]$ Return $\perp$
$\mathsf{st}_\mathsf{id}.\mathsf{InList} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{InList} + [\mathsf{in}]$

**Oracle** Send$(\mathsf{id}, \mathsf{m})$:
If id $\notin [1..k]$ Return $\perp$
If $\mathsf{st}_\mathsf{id}.\mathsf{stage} = 0$ :
    $(i, \mathsf{st}_\mathsf{id}.\mathsf{st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st}_\mathsf{id}.\mathsf{prms}, (p, (\mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)), \mathsf{in}_\mathsf{last}, \mathsf{m}, \mathsf{st}_V)$
    If $i = \perp$: Return $\perp$
    $(o, \mathsf{st}_\mathsf{id}.\mathsf{st}_L) \leftarrow\$ \mathsf{Loc}_\mathsf{KE}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L, i)$
    $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow o$
     If $\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}_\mathsf{id}.\mathsf{st}_L.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
        $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
        $\mathsf{st}.\mathsf{fake} \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
    If $(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{st}_\mathsf{KE}.\delta) = \mathsf{accept}$ : Then stage $\leftarrow 1$
    $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, o)$
    Return $\mathsf{m}'$
If $\mathsf{st}_\mathsf{id}.\mathsf{stage} = 1$ :
    If $\mathsf{m} = \epsilon$ :
        $\mathsf{in} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{InList}[0]$
        $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{ListIn}_\mathsf{id}$
        $\mathsf{st}_\mathsf{id}.\mathsf{ListIn}_\mathsf{id} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_{k-1})$
        $\mathsf{st}_\mathsf{id}.\mathsf{InList}[\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in}] \leftarrow \mathsf{in}$
        $o \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}), (\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in}, \{0\}^{|\mathsf{in}|}))$
        $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow o$
        $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} + 2$
        $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, o)$
        Return $\mathsf{m}'$
    Else:
        $\mathsf{m}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}), \mathsf{m})$
        If $\mathsf{m}' = (\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out}, \mathsf{out}')$ :
            $\mathsf{st}_\mathsf{id}.\mathsf{ListOut} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{OutList}[\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out}] : \mathsf{st}_\mathsf{id}.\mathsf{ListOut}$
            $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} + 2$
            $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)$
            Return $\mathsf{m}'$
Else: Return $\perp$

**Oracle** GetOutput$(\mathsf{id})$:
If id $\notin [1..k]$ Return $\perp$
$(\mathsf{out}_1, \ldots, \mathsf{out}_k) \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{ListOut}$
Return $\mathsf{out}_1 \,||\, \ldots \,||\, \mathsf{out}_i$
```

Fig. 22. Second hop of the proof.

```
$\mathbf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$(\mathsf{prms}, \mathsf{sk}) \leftarrow\$ \mathsf{SMInit}(1^\lambda)$
$\mathsf{hdl} \leftarrow 0$
$\mathsf{fake} \leftarrow [\,]$
$\mathsf{forgeAuth} \leftarrow \mathsf{F}$
$\mathsf{authList} \leftarrow [\,]$
$(\mathsf{st}_\mathcal{A}, k) \leftarrow\$ \mathcal{A}(\mathsf{prms})$
For $\mathsf{id} \in [1..k]$:
   $(\mathsf{st}_\mathsf{id}.\mathsf{st}_L, \mathsf{st}_\mathsf{id}.\mathsf{pub}) \leftarrow \mathsf{Setup}_\mathsf{KE}(1^\lambda, \mathsf{id})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_k)$
For $\mathsf{id} \in [k+1..n]$:
   $(\mathsf{st}_\mathcal{A}, \mathsf{pub}_\mathsf{id}) \leftarrow\$ \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{id}, \mathsf{Pub})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, \ldots, \mathsf{pub}_n)$
For $\mathsf{id} \in [1..k]$:
   $\mathsf{st}_\mathsf{id}.\mathsf{ListIn} \leftarrow [\,]; \mathsf{st}_\mathsf{id}.\mathsf{ListOut} \leftarrow [\,]; \mathsf{st}_\mathsf{id}.\mathsf{stage} \leftarrow 0$
   $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} \leftarrow 0; \mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} \leftarrow 1; \mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow \epsilon$
   $(\mathsf{Rem}^1_\mathsf{KE}, \ldots, \mathsf{Rem}^n_\mathsf{KE}) \leftarrow \mathsf{Pub}$
   $P \leftarrow \langle\, \langle \mathsf{Rem}^1_\mathsf{KE}, \ldots, \mathsf{Rem}^n_\mathsf{KE} \rangle_{1,\ldots,n}\, ; \, \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle \,\rangle_{\phi_\mathsf{key}, p, q}$
   $L \leftarrow \{(p, (\mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)), (q, \mathsf{st}_\mathsf{id}.\mathsf{id})\}$
   $\mathsf{st}_\mathsf{id}.\mathsf{st}_V \leftarrow (P, L)$
   $P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, P, L)$
$b \leftarrow\$ \mathcal{A}^\mathcal{O}(\mathsf{st}_\mathcal{A})$
If $\mathsf{forgeAuth} = \mathsf{T}$: $b \leftarrow\$ \{0, 1\}$

Oracle $\mathsf{Run}(\mathsf{hdl}, l, \mathsf{m})$:
$(P, \mathsf{seq}, \mathsf{st}_\mathsf{hdl}) \leftarrow \mathsf{HdlList}[\mathsf{hdl}]$
If $P = P^*$:/The agreed protocol.
   If $(p, (\mathsf{id}, \epsilon)) \notin L$: Return $\perp$
   If $\mathsf{st}_\mathsf{hdl}[\mathsf{id}].\mathsf{stage} = 0$ :
      $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{m}$
      $\mathsf{m}' \leftarrow\$ P^*[\mathsf{st}_\mathsf{hdl}, \mathsf{sk}](\mathsf{id}, \mathsf{m})$
      If $\mathsf{st}_\mathsf{id} \neq \epsilon \wedge \mathsf{st}_\mathsf{hdl}[\mathsf{id}].\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}_\mathsf{hdl}[\mathsf{id}].\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
         $\mathsf{key}^* \leftarrow\$ \{0, 1\}^{1^\lambda}$
         $\mathsf{st.fake} \leftarrow (\mathsf{st}_\mathsf{hdl}[\mathsf{id}].\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
   Else If $\mathsf{st}_\mathsf{hdl}[\mathsf{id}].\mathsf{stage} = 1$ :
      $(\mathsf{seq}_\mathsf{in}, \mathsf{id}, \mathsf{in}) \leftarrow \mathsf{m}$
      If $(\mathsf{seq}[\mathsf{id}] \neq \mathsf{seq}_\mathsf{in})$: Return $\perp$
      If $\mathsf{st}_\mathsf{id} \neq \epsilon$:/Honest participant
         If $\mathsf{m}' = (\mathsf{seq}[\mathsf{id}], \mathsf{out}')$ :
            $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_\mathsf{F}](\mathsf{id}, \mathsf{InList}[\mathsf{seq}[\mathsf{id}]])$
            $\mathsf{st}_\mathsf{id}.\mathsf{OutList}[\mathsf{seq}[\mathsf{id}] + 1] \leftarrow \mathsf{out}$
            $\mathsf{m}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_\mathsf{hdl}.\mathsf{key}[\mathsf{id}]), (\mathsf{seq}[\mathsf{id}] + 1, \{0\}^{|\mathsf{out}|}))$
            $\mathsf{authList} \leftarrow (\mathsf{fake}(\mathsf{st}_\mathsf{hdl}.\mathsf{key}[\mathsf{id}]), (\mathsf{seq}[\mathsf{id}] + 1, \{0\}^{|\mathsf{out}|})) : \mathsf{authList}$
      Else:/Corrupt participant
         $\mathsf{in}^* \leftarrow\$ \Lambda.\mathsf{Dec}(\mathsf{st}_\mathsf{hdl}.\mathsf{key}[\mathsf{id}], (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{in}))$
         $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_\mathsf{F}](\mathsf{id}, \mathsf{in})$
         $\mathsf{m}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{st}_\mathsf{hdl}.\mathsf{key}[\mathsf{id}], (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{out}))$
      $\mathsf{seq}[\mathsf{id}] \leftarrow \mathsf{seq}[\mathsf{id}] + 2$
Else:/Any other program on $\mathcal{M}$.
   $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{m}$
   $\mathsf{m}' \leftarrow\$ P^*[\mathsf{st}_\mathsf{hdl}, \mathsf{st.sk}](\mathsf{id}, \mathsf{m})$
$\mathsf{HdlList}[\mathsf{hdl}] \leftarrow (P, \mathsf{seq}, \mathsf{st}_\mathsf{hdl})$
Return $\mathsf{m}'$
```

```
Oracle $\mathsf{Load}(P)$:
$\mathsf{hdl} \leftarrow \mathsf{hdl} + 1$
For $i \in L$: $\mathsf{seq}[i] \leftarrow 0$
$\mathsf{HdlList} \leftarrow (\mathsf{hdl}, \mathsf{seq}, \epsilon)$
Return $\mathsf{hdl}$

Oracle $\mathsf{SetInput}(\mathsf{in}, \mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
$\mathsf{st}_\mathsf{id}.\mathsf{InList} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{InList} + [\mathsf{in}]$

Oracle $\mathsf{Send}(\mathsf{id}, \mathsf{m})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
If $\mathsf{st}_\mathsf{id}.\mathsf{stage} = 0$ :
   $(i, \mathsf{st}_\mathsf{id}.\mathsf{st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st}_\mathsf{id}.\mathsf{prms}, (p, (\mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)), \mathsf{in}_\mathsf{last}, \mathsf{m}, \mathsf{st}_V)$
   If $i = \perp$: Return $\perp$
   $(o, \mathsf{st}_\mathsf{id}.\mathsf{st}_L) \leftarrow\$ \mathsf{Loc}_\mathsf{KE}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L, i)$
   $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow o$
   If $\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}_\mathsf{id}.\mathsf{st}_L.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
      $\mathsf{key}^* \leftarrow\$ \{0, 1\}^\lambda$
      $\mathsf{st.fake} \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
   If $(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{st}_\mathsf{KE}.\delta) = \mathsf{accept}$ : Then stage $\leftarrow 1$
   $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, o)$
   Return $\mathsf{m}'$
If $\mathsf{st}_\mathsf{id}.\mathsf{stage} = 1$ :
   If $\mathsf{m} = \epsilon$ :
      $\mathsf{in} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{InList}[0]$
      $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{ListIn}_\mathsf{id}$
      $\mathsf{st}_\mathsf{id}.\mathsf{ListIn}_\mathsf{id} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_{k\text{-}1})$
      $\mathsf{st}_\mathsf{id}.\mathsf{InList}[\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in}] \leftarrow \mathsf{in}$
      $o \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}), (\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in}, \{0\}^{|\mathsf{in}|}))$
      $\mathsf{authList} \leftarrow (\mathsf{fake}(\mathsf{st}_\mathsf{hdl}.\mathsf{key}[\mathsf{id}]), (\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in}, \{0\}^{|\mathsf{in}|})) : \mathsf{authList}$
      $\mathsf{st}_\mathsf{id}.\mathsf{in}_\mathsf{last} \leftarrow o$
      $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{in} + 2$
      $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, o)$
      Return $\mathsf{m}'$
   Else:
      $\mathsf{m}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}), \mathsf{m})$
      If $\mathsf{m}' \neq \perp \wedge (\mathsf{fake}(\mathsf{st}_\mathsf{id}.\mathsf{st}_L.\mathsf{key}), \mathsf{m}) \notin \mathsf{authList}$:
         $\mathsf{forgeAuth} \leftarrow \mathsf{T}$
      If $\mathsf{m}' = (\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out}, \mathsf{out}')$ :
         $\mathsf{st}_\mathsf{id}.\mathsf{ListOut} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{OutList}[\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out}] : \mathsf{st}_\mathsf{id}.\mathsf{ListOut}$
         $\mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} \leftarrow \mathsf{st}_\mathsf{id}.\mathsf{seq}_\mathsf{out} + 2$
         $\mathsf{st}_\mathsf{id}.\mathsf{out} \leftarrow \mathsf{out}'$
         $\mathsf{m}' \leftarrow (\mathsf{st}_\mathsf{id}.\mathsf{stage}, \mathsf{st}_\mathsf{id}.\mathsf{id}, \epsilon)$
         Return $\mathsf{m}'$
Else: Return $\perp$

Oracle $\mathsf{GetOutput}(\mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
$(\mathsf{out}_1, \ldots, \mathsf{out}_k) \leftarrow \mathsf{ListOut}_\mathsf{id}$
Return $\mathsf{out}_1 \,||\, \ldots \,||\, \mathsf{out}_i$
```

Fig. 23. Third hop of the proof.