Foundations of Hardware-Based Attested Computation and Application to SGX*

Manuel Barbosa
HASLab, INESC TEC, FCUP
mbb@dcc.fc.up.pt

Bernardo Portela
HASLab, INESC TEC, UMinho
blfportela@gmail.com

Guillaume Scerri
University of Bristol
guillaume.scerri@bris.ac.uk

Bogdan Warinschi
University of Bristol
csxbw@bristol.ac.uk

Abstract

Exciting new capabilities of modern trusted hardware technologies allow for the execution of arbitrary code within environments completely isolated from the rest of the system and provide cryptographic mechanisms for securely reporting on these executions to remote parties.

Rigorously proving security of protocols that rely on this type of hardware faces two obstacles. The first is to develop models appropriate for the induced trust assumptions (e.g., what is the correct notion of a *party* when the peer one wishes to communicate with is a specific instance of an an outsourced program). The second is to develop scalable analysis methods, as the inherent stateful nature of the platforms precludes the application of existing modular analysis techniques that require high degrees of independence between the components.

We give the first steps in this direction by studying three cryptographic tools which have been commonly associated with this new generation of trusted hardware solutions. Specifically, we provide formal security definitions, generic constructions and security analysis for *attested computation*, *key-exchange for attestation* and *secure outsourced computation*. Our approach is incremental: each of the concepts relies on the previous ones according to an approach that is quasi-modular. For example we show how to build a secure outsourced computation scheme from an arbitrary attestation protocol combined together with a key-exchange and an encryption scheme.

1 Introduction

Background. The many applications that routinely manipulate sensitive data require strong guarantees which ensure i. that adversaries cannot tamper with their execution; and ii. that no sensitive information is leaked. Yet, satisfying these guarantees on modern execution platforms rife with vulnerabilities (e.g. in mobile devices, PCs) or inherently not trustworthy (e.g., cloud infrastructures) is a major challenge.

*This work was supported by the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE).

A promising starting point for solutions are the *remote attestation* capabilities offered by modern trusted hardware: computational platforms equipped with this technology can guarantee to a remote party various degrees of integrity for the software that it runs. For example the Trusted Platform Module (TPM) can provide certified measurements on the state of the platform and can be used to guarantee integrity of BIOS and boot code right before it is executed. More recent technologies (e.g., ARM's TrustZone and Intel's Software Guard Extension (SGX) [18]) have significantly expanded the scope and guarantees of trusted hardware. They offer the ability to run applications in "clean-slate" isolated execution environments (IEE) completely independent of anything else running on the processor; the desired attestation guarantees come from reports that are authenticated cryptographically.

A second major challenge is to provide security guarantees that go beyond heuristic arguments. Here, the established methodology is the "provable security" approach, which advocates carrying out the analysis of systems with respect to rigorously specified models that clarify the trust relations, the powers of the adversary and what constitutes a security breach. The approach offers well-established definitional paradigms for all basic primitives and some of the more used protocols. On the other hand, the success of applying this approach to new and more complex scenarios fundamentally hinges on one's ability to tame scalability problems, as models and proofs, even for moderate size systems, tend to be unwieldy. Some solutions to this issue exist in the form of compositional principles that, when incorporated as a native feature in the security abstractions, allow to establish the guarantees of larger systems from the guarantees on its components [8].

In this paper we take a provable security approach to protocols that rely on the IEE-capabilities of modern trusted hardware. It may be tempting to assume that, for the analysis of such protocols, designing security models is a simple matter of overlaying/merging the trust model induced by the use of such hardware over well-established security abstractions. If this were true, one could rely on established models and methodologies to perform the analysis. Unfortunately, this is not the case.

Consider the problem of secure outsourced computation of a program P to a remote machine. The owner of P wants to ensure that the (potentially malicious) remote platform does not tamper with the execution of the program and that it learns no information about the input/output (I/O) behavior of P. For remote machines with IEE capabilities, the following straightforward design has been informally proposed in the literature and should, intuitively, provide the desired guarantees. First, execute a key-exchange with a remote instance of an IEE; after the key exchange finishes, use the key thus established to send encrypted inputs to the IEE who can decrypt and pass them to P. The output returned by P is encrypted within the IEE and sent to the user. The construction relies on standard building blocks (key-exchange, authenticated encryption) so the security of the overall design should be reducible to that of the key exchange and authenticated encryption, for which we already have widely-accepted security models and constructions.

We highlight two important issues that show that neither existent models nor existent techniques are immediately suitable for the analysis of IEE-based protocols in general, and for the protocol above in particular. The first is the concept of a *party* which is a key notion in specifying and reasoning about the security of distributed systems. Traditionally, one considers security (e.g., of a key-exchange) in a setting where there is a PKI and at least some of the parties (e.g., the servers) have associated public keys. Parties and their cryptographic material are then essentially the same thing for the purpose of the security analysis. In the context that we study, users (who wish to use trusted hardware) are not expected to have

long term keys; furthermore, privacy considerations require that the cryptographic operations performed by the trusted hardware on remote machines should not allow one to track different instances – which makes long term cryptographic material inadequate as a technical anchor of a party's participation in such a protocol. Indeed, the desirable functionality for many usages of such systems is that the cryptographic material associated with a computation outsourcing protocol (both for local and remote *parties*) can be arbitrary and fixed on-the-fly, when the protocol is executed. An interesting problem is therefore how to define the security of outsourced computation in this setting, and how to rely on the asymmetry afforded by the trust model specific to IEE systems to realise it.

The second issue is *composability*. In the protocol for outsourced computation that we present above, one might be led to think that security simply follows if the key-exchange is secure and the channel between the user and the IEE uses authenticated encryption (with appropriate replay protection): if the only information passed from the key exchange to the channel is the encryption key, then one can design and analyze the two parts separately. While in more standard scenarios this may be true, reliance on the IEE breaks the independence assumption that allows for composability results: what the specification above hides is that the code run by the IEE (i.e., the program for key-exchange, the one for the secure channels and the program that they protect) needs to be loaded at once, or else no isolation guarantees are given by the trusted hardware.[1] This means that the execution of the different parts of the program is not necessarily independent, as they unavoidably share the state of the IEE. An important question is therefore whether the above intuitive construction is sound, and under which conditions can one use it to perform IEE-enabled outsourcing of computation.

To summarise the above discussion, two remarks help motivate the work in this paper. Protocols relying on IEE are likely to be deployed in applications with stringent security requirements, so ensuring that they fall under the scope of the provable security approach is important. Moreover, such applications are likely to be complex: inherently, they involve communication between remote parties, incorporate diverse code executed at different levels of trust, and rely on multiple cryptographic primitives and protocols as building blocks (see, for example, the One-time password protocol based on SGX[18]). However, we have concluded that key aspects of existing cryptographic models do not naturally translate to this new setting and, perhaps more worryingly, that the type of compositional reasoning enabled by such cryptographic models is clearly unsuitable for protocols that rely on IEE.

Our approach. We will first outline the high-level decisions that underly our approach, and then describe our technical contributions more in detail. We use SGX and TrustZone as inspiration, but we do not hardwire our models to a specific platform, in order to ensure that the scope of our work encompasses other similar technologies. Instead, we use an abstract notion of a machine which captures the relevant aspects that such platforms offer: their capability to run processes with isolation guarantees and the ability to directly use secure

[1] Intuitively, mechanisms such as SGX and TrustZone are designed to protect and provide attestation guarantees over monolithic pieces of software, which must be fixed when an IEE is created and are identified using a fingerprint of the code. To go around this restriction and compose multiple programs, one has two options: i. to build a single composed program and load it in its entirety into an IEE (this is the approach we follow in this paper); or ii. to use multiple IEEs to host the various programs, and employ cryptographic protocols to protect the interactions between them using the (potentially malicious) host operating system as a communications channel. We note that this latter option would again lead to loading composed programs into each IEE, since cryptographic code would need to be added to each of the individual programs, and this would then lead to the same problem that we intend to solve with the framework we propose in this paper.

cryptography without going through potentially untrusted software. Additionally, we target our effort on the combination of remote attestation and key-exchange protocols. The former is *the* raison d'être for trusted hardware, while the latter is the natural building block towards adding secrecy guarantees for code running under the protection of IEEs (per our example above).

As explained above, one challenge we set out to address is to incorporate into our approach a new form of compositional reasoning that permits dealing with potentially shared state between all the code that is loaded into an IEE, which means that a-priori there are no guarantees of independence between the executions of different cryptographic primitives that one could incorporate into the same program. Indeed, at the very least, the reporting mechanisms for the IEE will refer to the code of the full program, which immediately constrains modular reasoning — a crucial tool to enable scalability. We sidestep this problem by providing definitions (for both syntax and security) that are *composition-aware*: they explicitly assume that the code loaded into an IEE may result from the composition of multiple programs.

Contributions

We focus on the interplay between composition and attestation in hardware-based settings. In particular, we concentrate on a pervasive use case: attestation is often used to protect a security-critical part of the code (P^* in our setting) whereas the remaining code Q does not need attestation: it can rely on guarantees established by P^* (e.g. an authenticated secret key), and can therefore be much more efficient. As explained above and in footnote 1 the stateful nature of the execution environment does not allow for independent analysis of these two components and new techniques for rigorous validation are needed.

Our starting point is therefore a program Q that a local user wishes to outsource to a remote machine. Such a program will need to be transformed (*compiled* in the cryptographic sense of the word) into another program that, intuitively, will result from the composition of a handshake/bootstrapping procedure P^* that will establish a secure channel with the IEE in which program Q will be executed, and an instrumented version of Q, say Q^*, that uses the aforementioned secure channel to ensure that Q is indeed securely executed.

Our approach to formalising and realising this composition pattern has three main stepping-stones: i. we introduce *attested computation* as the formalisation of the raw guarantees provided by IEEs with cryptographic functionalities; ii. we show how a passively secure key exchange can be efficiently combined with an *attested computation* scheme to obtain the bootstrapping procedure P^* referred above; and iii. we rely on our composition-aware formalisation to show that by instrumenting program Q^* using standard cryptographic techniques one achieves secure outsourced computation. Details follow.

Attested computation. Our first contribution is a formal treatment of IEE-based remote attested computation. We consider a setting where a user wishes to remotely execute a program P and rely on the cryptographic infrastructure available within the IEE to attest that some incarnation P^* of this program is indeed executing within an IEE of a specific remote machine (or group of machines). We provide a general solution to this problem in the form of a new cryptographic concept called an *attested computation*. We formalize two core guarantees.

First, we demand that the user's local view of the execution is "as expected", i.e., that the I/O behavior that is reconstructed locally corresponds to an honest execution of P. The

second guarantee is more subtle and requires that such an *execution has actually occurred in an IEE within a specific remote platform*; in other words, the attested computation client is given the assurance that its code is being run in isolation (and displays a given I/O behaviour) within a prescribed remote physical machine (or group of machines) associated with some authenticated public parameters. This latter guarantee is crucial for bootstrapping the secure outsourcing of code: consider for example P to be a key-exchange protocol, which will be followed by some other program that relies on the derived key. It must be the case that, at the end of the key exchange, the remote state of the key-exchange is protected by an IEE.

The second guarantee we demand from attested computation follows easily from the previous observation. If the attested program keeps sensitive information in its internal state (which is not revealed by its I/O behaviour as in the case of a key exchange protocol) then execution within the remote IEE should safeguard its internal state. If this were not the case, we would again run into problems when trying to compose P^* with some program that relies on the security properties of P. We therefore exclude attested computation schemes where the instrumented program P^* might leak more information in its I/O behaviour than P itself by introducing the notion of *minimal leakage*, which essentially states that the I/O of an attested program does not leak any information beyond what is unavoidably leaked by an honest execution.

Finally, we provide a scheme for attested computation that relies on a remote machine offering a combination of symmetric authentication and digital signatures (a capability similar to what SGX provides) and show that our scheme is secure in the sense that we define.

Key-exchange for attested computation. On its own, attested computation only provides integrity guarantees: the I/O behavior of the outsourced code is exposed to untrusted code in the remote machine on which it is run. The natural solution to the problem is to establish a secure communication channel with the IEE via a key-exchange protocol. It is unclear, however, how the standard security models for key exchange protocols map into the attested computation scenario, and how existing constructions for secure key-exchange fare in the novel scenario that we study. Indeed, for efficiency reasons one should use a key exchange protocol that is *just strong enough* to achieve this goal.

To clarify this issue we formalize the notion of key-exchange *for* attested computation. The name that we propose is intentional: key-exchange protocols as used in the our context differ significantly in the syntax and security models from their more traditional counterparts. For example, our syntax reflects that the code of the key-exchange is not fixed a-priori: a user can set parameters both for the component to be run locally and for the one to be executed within the IEE. This allows a user to hardwire in the code to be run remotely a new nonce (or as in our examples some cryptographic public key for which it knows the secret key).

As explained above, the notion of *party* in the context of attested computation needs to be different from that adopted by traditional notions of secure key-exchange. Our solution is to rely on the trust model specific to IEE settings: we can assign some arbitrary strings as identifiers for the users of the local machine, and we allow these users to specify arbitrary strings as identifiers for the remote code (a secure instantiation would require that this identifier corresponds to some cryptographic material possibly generated on the fly as explained above). We then adapt the execution model and definitions for key-exchange for the modified syntax and the new notion of communicating parties to reflect the expected guarantees: different local and remote sessions agree on each other's identifiers, derive the same key and the key is unknown to the adversary. One crucial aspect of our security model for key-exchange

is that it explicitly accounts for the fact that the remote process will be run under attestation guarantees, which maps to a *semi-active* adversarial environment.

To improve usability of our notion of secure key-exchange for attested computation we provide two results. The first result simplifies the design of such protocols. Here, we show a generic construction that combines a key-exchange protocol that is passively secure and a standard signature scheme to derive a (potentially very efficient) key exchange protocol for attestation. The second result simplifies reasoning about the composition of a key-exchange for attestation with an arbitrary protocol that relies on the agreed key. Specifically, we provide a *utility theorem* which specifically states what composition guarantees one gets for an arbitrary program Q that is run within a remote IEE and relies on a shared key that was established via the attested computation of a key exchange protocol that satisfies our tailored definition.

Secure outsourced computation. The last layer in our framework is a formalisation of secure outsourced computation, the principal motivating use-case for our approach. We provide syntax and two security notions for a secure outsourced computation protocol, one for authenticity and the second one for the privacy of the I/O of the outsourced program. We then prove that the construction that combines a key-exchange for attested computation with an authenticated symmetric encryption scheme and replay protection gives rise to a scheme for secure outsourced computation. We present our result as a general formalisation (i.e., not application specific) of the intuition that by relying on more powerful hardware assumption such as those offered by SGX, one can indeed efficiently achieve a well-defined notion of secure outsourced computation that simultaneously offers verifiability *and* privacy. The proof of this result crucially relies on the utility theorem we defined for the combination of attested computation with key exchange.

2 Other Related work

Work that looks at provable security of realistic protocols that use trusted hardware-based protocols has developed around the protocols offered by the Trusted Platform Module (TPM) [5, 28, 6, 12, 11]. However, the functionality and efficiency of the protocols offered by the TPM makes them more suitable for static attestation (i.e., ensuring integrity of programs right before they are executed). Run-time guarantees, like those that we study here are in principle possible but cumbersome to obtain. Linking attestation guarantees provided by the TPM with those offered by a secure channel onto a remote machine had been studied before but only informally [16]. Although more rigorous approaches used to analyze attestation guarantees for protocols based on the TPM exist, they use more abstract models (with weaker guarantees) [27, 10].

Another related line of research leverages the trusted hardware to bootstrap entire platforms for secure software execution (e.g. Flicker [22], Trusted Virtual Domains [9], Haven [2]). These are large systems that are currently outside the scope of provable-security techniques. Smaller protocols which solve specific problems (secure disk encryption [23], one-time password authentication [18] outsourced Map-Reduce computations [26], Secure Virtual Disk Images [13], secure embedded devices [24, 21]) are more susceptible to rigorous analysis. Although some protocols (e.g., those of Hoekstra et al. [18]) come only with intuition regarding their security, others – most notably those by Schuster et. al [26] which uses SGX platforms to outsource map-reduce computation – come with a proof of security. The constructions in

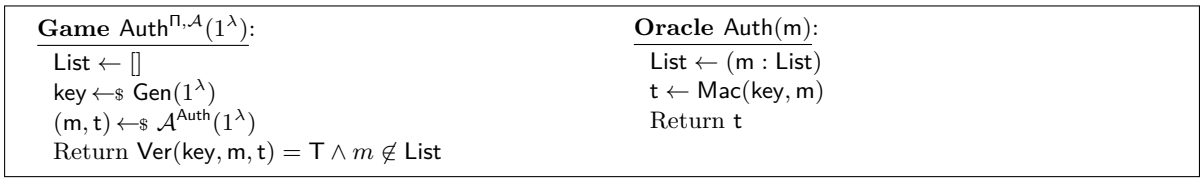

Figure 1: Game defining the security of a MAC scheme Π.

that paper are close to those that we abstract and analyze here.

Our construction of secure outsourced computation can be seen as a solution to the problem of verifiable computation as specified by [14]. That rich line of work concentrates on the much harder problem of providing crypto-only solutions and usually gives up privacy for efficiency and verifiability (e.g., [4], [15], [25]). Here, we show how to obtain a reasonably efficient and secure system with technology that is likely to be deployed in the near future.

3 Preliminaries

3.1 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 [3]. 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 1 returns T.

3.2 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 [17] . 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 2 returns T.

3.3 Passively secure key exchange

We define a form of key exchange protocol that does not rely on long term secret/state or global setup and for which we require weak security guarantees (essentially security against a passive adversary); the classical Diffie-Hellman key exchange is a standard example. Later we

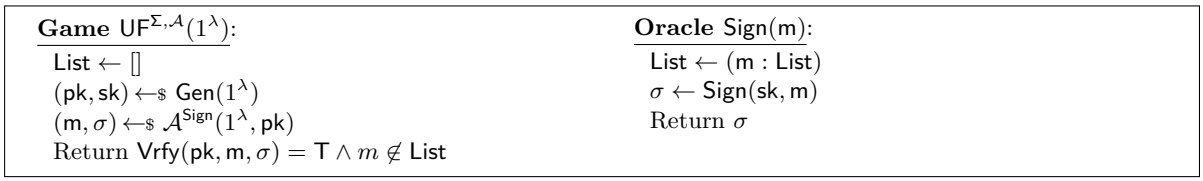

Figure 2: Game defining the security of a signature scheme Σ.

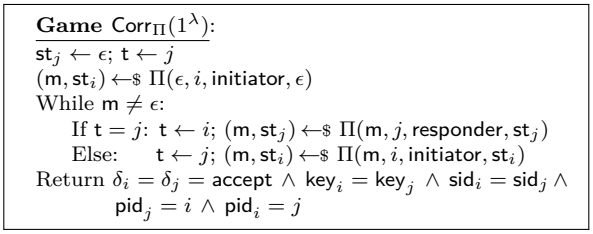

Figure 3: Game defining the correctness of protocol Π.

show how, combined with attestation, such protocols yield secure key-exchange when facing active adversaries. A key exchange protocol is therefore defined by a single ppt algorithm Π used by communicating parties. When analysing the security and correctness of the protocol we will consider that each party with identifier id can execute several instances of the protocol with different parties. Throughout the paper we let identifiers be arbitrary strings, which will be given meaning by the higher-level application relying on the protocol under analysis. For $s \in \mathbb{N}$, we write Π^s_{id} for the s instance of party id. We assume that each instance maintains variables $\mathsf{st}, \delta, \mathsf{key}$ which record respectively, local state information, the state of the key ($\mathsf{derived}, \mathsf{accept}, \mathsf{reject}$ or $\perp$) and the value of the key. In addition, we assume variables for the role ρ of the session (initiator or responder), the party identifier of the owner of the session oid and that of its partner pid and a session identifier sid. We require that $\mathsf{key} = \perp$ unless $\delta \in \{\mathsf{derived}, \mathsf{accept}\}$, that oid, ρ, sid, key are only assigned once during the entire execution of the protocol (the first two when the session is initialized). We denote running Π with message m, role ρ and state st to produce m' and the updated state st' by $(\mathsf{m}', \mathsf{st}') \leftarrow\$ \Pi(1^\lambda, \mathsf{m}, \mathsf{id}, \rho, \mathsf{st})$, and will omit the security parameter input throughout the paper for the sake of compactness. We will use message ϵ to refer to the empty string. This will be passed to both parties as the initial state; it will be passed to the initiator as first input message; and it will be returned as output message by the party that executes last, to denote that no further interaction is needed. Long term secrets and/or shared initial state between several instances run by the same identity can be captured by setting their initial state accordingly. Correctness. A key exchange protocol is correct if, after a complete (honest) run between two participants with complementary roles, both reach the accept state, both derive the same key and session identifier, and both obtain correct partner identities. More formally, a protocol Π is correct if, for any distinct party identities i and j, the experiment in Figure 3 always returns T.

Execution model. We will consider key exchange schemes that are secure against passive adversaries. Our adopted security notion is a restriction of the scenario considered in [20] that excludes corruptions.[2] The execution model considers an adversary, which is run on

[2]The trust model we consider for attested computation excludes corruptions for the sake of simplicity, but all our results can be extended to consider that possibility. Having said that, it seems reasonable to exclude the possibility of isolated executed environment breach based on a hardware assumption. The possibility of

the security parameter, and which can interact with the following oracles whose behaviour depends on a secret sampled bit b and a shared list of pairs of keys fake, which is initially empty:

- $\mathsf{Execute}(i, j)$ runs a new instance of the protocol between distinct parties i and j. It then checks if the key derived for the executed session exists in list fake. If not, it generates a new key^* uniformly at random, and adds $(\mathsf{key}, \mathsf{key}^*)$ to the list. Finally, it outputs the transcript of the protocol execution and the session identifier associated with it.
- $\mathsf{Reveal}(i, s)$ outputs the session key key of Π_i^s.
- $\mathsf{Test}(i, s)$ will return $\perp$ if $\delta_i^s \neq \mathsf{accept}$ (i.e. if no execute query actually created such a session). Otherwise, if $b = 0$ it outputs the key associated with Π_i^s. If $b = 1$, it searches for the key associated with Π_i^s in list fake and returns the associated key^*.

When the adversary terminates interacting with the oracles, it will eventually output a bit b' which represents his guess on what the challenge bit b is.

Partnering relations. We define entity authentication following [7], and observe that in the case of passively secure key exchange, this is essentially a correctness property. First we introduce a notion of partnering, which informally states that two oracles which have derived keys are partners if they share the same session identifier. The definition makes use of the following predicate on two instances Π_i^s and Π_j^t holding states $(\mathsf{st}_i^s, \delta_i^s, \rho_i, \mathsf{sid}_i^s, \mathsf{pid}_i^s, \mathsf{key}_i^s)$ and $(\mathsf{st}_j^t, \delta_j^t, \rho_j, \mathsf{sid}_j^t, \mathsf{pid}_j^t, \mathsf{key}_j^t)$, respectively:

$$\mathrm{P}(\Pi_i^s, \Pi_j^t) = \begin{cases} \mathsf{T} \text{ if } \mathsf{sid}_i^s = \mathsf{sid}_j^t \ \wedge \ \delta_i^s, \delta_j^t \in \{\mathsf{derived}, \mathsf{accept}\} \\ \mathsf{F} \text{ otherwise.} \end{cases}$$

Definition 1. [Partner]Two players Π_i^s and Π_j^t are partnered if $\mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{T}$.

Our authentication notion relies on three further definitions, which demand that partnerings will need to be *valid*, *confirmed* and *unique*. In short, these three requirements ensure that any instance that accepts has a partner, that this partner is unique and that partners share the same key.

Definition 2 (Valid Partners)**.** A protocol Π ensures valid partners if the bad event notval does not occur, where notval is defined as follows:

$$\begin{aligned} &\exists \Pi_i^s, \Pi_j^t \ \text{ s.t. } \mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{T} \wedge \\ &\quad (\, \mathsf{pid}_i^s \neq \mathsf{oid}_j^t \ \vee \ \mathsf{pid}_j^t \neq \mathsf{oid}_i^s \ \vee \ \rho_i = \rho_j \ \vee \ \mathsf{key}_i^s \neq \mathsf{key}_j^t \,). \end{aligned}$$

Definition 3 (Confirmed Partners)**.** A protocol Π ensures confirmed partners if the bad event $\mathsf{notconf}$ does not occur, where $\mathsf{notconf}$ is defined as follows:

$$\exists \Pi_i^s \ \text{ s.t. } \ \delta_i^s = \mathsf{accept} \wedge \forall \Pi_j^t, \mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{F}.$$

Definition 4 (Unique Partners)**.** A protocol Π ensures unique partners if the bad event notuni does not occur, where notuni is defined as follows:

$$\begin{aligned} &\exists \Pi_i^s, \Pi_j^t, \Pi_k^r \ \text{ s.t.} \\ &\quad (j, \mathsf{t}) \neq (k, r) \ \wedge \ \mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{T} \ \wedge \ \mathrm{P}(\Pi_i^s, \Pi_k^r) = \mathsf{T} \end{aligned}$$

local machine corruption should, however, be considered.

Intuitively, we will consider that an adversary violates two-sided entity authentication if he can lead an instance of an honest party running the protocol to accept, and in doing that cause one of the bad events notval, notconf, notuni.

Security definition. We are now ready to present the security notion we will be using for key exchange protocols. To exclude breaks via trivial attacks, we define legitimate adversaries as those who ensure the following freshness criteria is satisfied for his $\mathsf{Test}(i,s)$ queries: i. $\mathsf{Reveal}(i,s)$ was not queried; and ii. for all Π_j^t such that $\mathrm{P}(\Pi_i^s,\Pi_j^t)=\mathsf{T}$, $\mathsf{Reveal}(j,t)$ was not queried. We only consider experiments in which the adversary is found to be legitimate and define the winning event guess to be $b=b'$ at the end of the experiment.

Definition 5. [Passive AKE security] A protocol Π is passively secure if, for any legitimate ppt adversaryinteracting with the execution environment described above: 1. the adversary violates two-sided entity authentication with negligible probability$\Pr[\,\mathsf{notval}\vee\mathsf{notconf}\vee\mathsf{notuni}\,]$; and 2. its key secrecy advantage $2\cdot\Pr[\,\mathsf{guess}\,]-1$ is negligible.

One-Sided Authentication. We will also consider a weaker form of entity authentication guarantee, where only some parties are authenticated. To this end, we will distinguish between Loc and Rem parties, identify the former with responders, and the latter with initiators. In one-sided authentication, only initiators (i.e., remote parties) are authenticated, which means that responders (i.e., local parties) do not need to keep any long term secrets. Again, we follow [7] and give one-sided versions of the definitions for valid partners and confirmed partners. Intuitively, on acceptance, a local party will be assured that a valid unique partnering session exists. Security will still need to ensure that each local (or remote) session has at most one remote (respectively local) partnered session. However, we no longer confirmation for the remote machine: we allow the remote party to accept even if there is no local matching session that has accepted. This weaker guarantee is sufficient for many applications and has been used in the context of attested computation, for example, in [18].

Definition 6 (One-Sided Valid Partners)**.** A protocol Π ensures one-sided valid partners if the bad event os-notval does not occur, where os-notval is defined as follows:

$$\exists\Pi_i^s,\Pi_j^t \;\text{ s.t. }\; i\in\mathsf{Loc}\wedge\mathrm{P}(\Pi_i^s,\Pi_j^t)=\mathsf{T}\;\wedge\\ (\,\mathsf{pid}_i^s\neq j\;\vee\;\rho_i^s\neq\mathsf{initiator}\;\vee\;\rho_j\neq\mathsf{responder}\;\vee\;\mathsf{key}_i^s\neq\mathsf{key}_j^t\,).$$

Definition 7 (One-Sided Confirmed Partners)**.** A protocol Π ensures one-sided confirmed partners if the bad event os-notconf does not occur, where os-notconf is defined as follows:

$$\exists\Pi_i^s \;\text{ s.t. }\; i\in\mathsf{Loc}\wedge\\ \delta_i^s=\mathsf{accept}\wedge\forall\Pi_j^t,\,\mathrm{P}(\Pi_i^s,\Pi_j^t)=\mathsf{F}.$$

When one-sided authentication suffices, then the definition of key exchanged security is weakened by relaxing condition 1 in Definition 5, which is modified to refer to the following event.

$$\Pr[\,\mathsf{os\text{-}notval}\vee\mathsf{os\text{-}notconf}\vee\mathsf{notuni}\,]$$

4 IEEs, Programs, and Machines

Isolated Execution Environments. At the high-level, an IEE can be seen as an idealised random access machine running some fixed program P, whose behaviour can only be influenced via a well-specified interface that permits passing inputs to the program, and receiving

its outputs. Intuitively, an IEE gives the following security guarantees, which we will formalise later in this section. The I/O behaviour of a process running in an IEE is determined by the program it is running, the semantics of the language in which the program is written, and the inputs it receives. This means, in particular, that there is strict isolation between processes running in different IEEs (and any other program running on the machine). Furthermore, the only information that is revealed about a program running within an IEE is contained in its input-output behaviour (which in most hardware systems is simply shared memory between the protected code and the untrusted software outside).

We emphasize that our notion of a machine is intended to be inclusive of any hardware platform that supports some form of isolated execution. For this reason, the syntax of this abstraction is minimalistic, so that it can be restricted/extended to capture the specific guarantees awarded by different concrete hardware architectures, including TPM, TrustZone, SGX, etc. As an example, our "vanilla" machine supports an arbitrary number of IEEs, where programs can be loaded only once, and where multiple input/output interactions are allowed with the protected code. This is a close match to the SGX/TrustZone functionalities. However, for something like TPM, one could consider a restricted machine where a limited number of IEEs exist, with constrained input/output capabilities, and running specific code (e.g., to provide key storage). Similarly, we consider IEE environments where the underlying hardware is assumed to only keep *benevolent* state, i.e., state that cannot be used to introduce destructive correlations between multiple interactions with an IEE. Again, this closely matches what happens in SGX/Trustzone, but different types of state keeping could be allowed for scenarios where such correlations are not a problem or where they must be dealt with explicitly.

PROGRAMS. Implicit throughout the paper will be a programming language $\mathcal{L}$ in which programs are written. We assume that this language is used by all computational platforms, but we admit IEE-specific system calls giving access to different cryptographic functionalities. These are referred as the *security module* interface. An additional system call rand is also assumed to be present in all platforms, giving access to fresh random coins sampled uniformly at random. Language $\mathcal{L}$ is assumed to be deterministic modulo the operation of system calls. As mentioned above, 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.

A program P must be written as a transition function, mapping bit-strings to bit-strings. Such functions take a current state st and an input i, and they will produce a new output o and an updated state. We will refer to this as an *activation* and express it as $o \leftarrow P[\mathsf{st}](i)$. Unless otherwise stated, st will be 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. The transition function may return arbitrary output until it produces an output where $\mathsf{finished} = \mathsf{T}$, at which point it can return no further output or change its state. We extend our notation as $o \leftarrow P[\mathsf{st}; r](i)$ to account for the randomness

obtained via the rand system call as extra input r; and as $(o_1, \ldots, o_n) \leftarrow P[\mathsf{st}; r](i_1, \ldots, i_n)$ to represent a sequence of activations. We write $\mathsf{Trace}_{P[\mathsf{st};r]}(i_1, \ldots, i_n)$ for the corresponding I/O trace $(i_1, o_1, \ldots, i_n, o_n)$.

Program composition. Given two programs P and Q, and a projection function between the internal states of the two programs ϕ, we will refer to the sequential composition of the two programs as $\mathsf{Compose}_{\phi}\langle P, Q\rangle$. This is defined as a transition function R that has two execution stages, which are signaled in its output via an additional stage bit. In the first stage, every input to R will activate 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 R 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 (termination being counted as a third stage) is appended to any output of a composition. When dealing with such a composed program, we will denote by $\mathsf{ATrace}_{R[\mathsf{st};r]}(i_1, \ldots, i_n)$ the prefix of the trace that corresponds to the execution of P. Intuitively, this denotes the *attested trace* where only the initial part of the program must be protected via attestation.

Machines. A *machine* $\mathcal{M}$ is an abstract computational device that captures the resources offered by a real world computer or group of computers, whose hardware security functionalities are initialised by a specific manufacturer before being deployed, possibly in different end-users.For example, a machine may represent a single computer produced by a manufacturer, configured with a secret signing key for a public key signature scheme, and whose public key is authenticated via some public key infrastructure, possibly managed by the manufacturer itself. Similarly, a machine may represent a group of computers, each configured with secret signing keys associated with a group signature scheme; again, the public parameters for the group would then be authenticated by some appropriate infrastructure.[3] In this paper we will restrict ourselves to the simplest case where standard public key signatures are used; but all our results easily extend to more complex group management schemes.

We will 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. Loosely speaking, this interface can be thought of as the ideal functionality that captures a system such as SGX [19]. The interface is as follows:

- $\mathsf{Init}(1^{\lambda})$ 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. We emphasize that the global parameters of machines are the only pieces of information that are assumed to be authenticated using external mechanisms such as a PKI in the entire paper.

- $\mathsf{Load}(P)$ 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 not to its randomness input. This means that

[3] If the possibility of removing elements from the group is not needed, then even sharing the same signing key for a public key encryption scheme between multiple computers could be a possibility.

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

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

We define the I/O trace $\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl})$ of a process hdl running in some machine $\mathcal{M}$ as the tuple $(i_1, o_1, \ldots, i_n, o_n)$ that includes the entire sequence of n inputs/outputs resulting from all invocations of the Run procedure on hdl; $\mathsf{Program}_{\mathcal{M}}(\mathsf{hdl})$ is the code (program) running inside the process with 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 denote by $\mathcal{A}^{\mathcal{M}}$ the interaction of some algorithm with a machine $\mathcal{M}$, i.e., having access to the Load and Run oracles defined above.

Remark. Here, we use hdl as a convenient identifier for the secure environment executing process P; in some incarnation the handle could be defined as a tuple containing the identity of the machine, some identifier for the secure environment and, say, the hash of the program P. More detailed formalisms are possible. We may consider, for example, different entry/exit points related to P. We may also explicitly refine P as a program and some initial associated data.

5 Attested Computation

We now formalise a cryptographic primitive that aims to address the remote execution, i.e., outsourcing, of programs as illustrated in Figure 4. In this setting, a user running software in a trusted local machine wishes to use an untrusted network to access a pool of remote machines with IEE facilities. The remote machines will be running general-purpose operating systems and other untrusted software. The goal of the user is to run a specific program P within an IEE in one of the remote machines, and to obtain assurance that, not only the program is indeed executing there, but also that it is displaying a particular I/O behaviour.We call this *attested computation*, and introduce it as the cryptographic primitive that formalises the simplest cryptographic application of trusted hardware systems offering IEE functionality, such as Intel's SGX architecture. Attested computation will be the lowest of a series of cryptographic layers that we will be building on top of each other throughout the paper, until eventually we show how such hardware assumptions permit efficiently realising a strong form of secure verifiable computation.

Syntax. An *Attested Computation* (AC) scheme is defined by the following algorithms:

- $\mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$ is the program compilation algorithm. On input global parameters for some machine $\mathcal{M}_R$, and programs P and Q, whose composition under projection function ϕ will be outsourced, it will output program R^*, together with an initial (possibly empty) state st for the verification algorithm. This algorithm is run locally. R^* is the code to be run as an isolated process in the remote machine. Intuitively, P is the initial part of the remote code that requires attestation guarantees, whereas Q is any subsequent code that may be remotely executed (generally leveraging the security guarantees that have been bootstrapped using the initial attested execution).

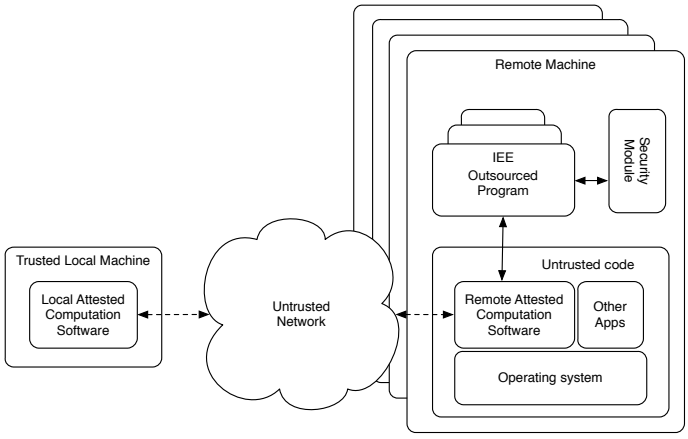

Figure 4: Attested Computation scenario.

- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i)$ is the attestation algorithm. On input global parameters for $\mathcal{M}_R$, a process handle hdl and an input i, it will use the interface of $\mathcal{M}_R$ 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 R^*, providing it with inputs and recovering the (possibly attested) outputs that should be returned to the local machine.

- $\mathsf{Verify}(\mathsf{prms}, i, o^*, \mathsf{st})$ is the (stateful) output verification algorithm. On input global parameters for $\mathcal{M}_R$, an input i, a (possibly attested) output o^* and some state st, it will produce 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.

In Figure 4, the local attested computation software block corresponds to $\mathsf{Compile}$ (one initial usage per program) and Verify (one usage per incoming attested output), whereas the remote attested computation software block corresponds to Attest (one usage per remote program activation, i.e. per I/O transition). The above syntax can be naturally extended to accommodate the simultaneous compilation of multiple input programs and/or the possibility that $\mathsf{Compile}$ may generate multiple output programs. This would allow us to capture, e.g., map/reduce applications such as those described in [26].

Correctness. Intuitively, an AC scheme is correct if, for any given programs P and Q and assuming an honest execution of all components in the scheme, both locally and remotely, the local user is able to accurately reconstruct a view of the I/O sequence that took place in the remote environment. Furthermore, this I/O sequence must be consistent with the semantics of $\mathsf{Compose}_\phi\langle P; Q\rangle$. In other words, suppose the compiled program is run under handle hdl^* in remote machine $\mathcal{M}_R$, and the local user uses Verify to reconstruct the remote I/O behaviour $(i_1, o_1, \ldots, i_n, o_n)$. Then, if we define $R := \mathsf{Compose}_\phi\langle P; Q\rangle$, we must have

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

The following definition formalizes the notion of a local user *correctly remotely executing program* P using attested computation.

Definition 8. [Correctness] An Attested Computation scheme AC is correct if, for all λ, and all adversaries $\mathcal{A}$, the experiment in Figure 5 (top) 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 introduced next.

Structural preservation. Since we are dealing with composed programs, we extend the correctness requirements on attested computation schemes to preserve the structure of the input program (P, ϕ, Q), and to modify only the part of the code that will be attested. Formally, we impose that, given any program P, there exists a (unique) compiled program P^*, such that, for any mapping function ϕ and any program Q, we have that $\mathsf{Compose}_{\phi}\langle P^*; Q\rangle = \mathsf{Compile}(P, \phi, Q)$.

Security. Security of an attested computation scheme imposes that an adversary with absolute control of the remote machine cannot convince the local user that some arbitrary remote execution of a program P has occurred, when it has not (nothing is said about the subsequent remote execution of program Q). Formally, we allow the adversary to freely interact with the remote machine, whilst providing a sequence of (potentially forged) attested outputs. 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. the execution trace that is validated by Verify is inconsistent with the semantics of P (in which case an adversary would be able to convince the local user of an I/O sequence that could not possibly have occurred!); or ii. there does not exist a remote process hdl^* exhibiting a consistent execution trace (in which case, the adversary would be able to convince the local user that a process running P was executing in the remote machine, when it was not).

Since the adversary is free to interact with the remote machine as it pleases, we can not hope to prevent it from appending arbitrary inputs to the trace of any remote process, while refusing to deliver all of the resulting attested outputs to the local user. This justifies the winning condition in our security game referring to a prefix of the trace in the remote machine, rather than imposing trace equality. Indeed, the definition's essence is to impose that the locally recovered trace and the remote trace share a common prefix ($\sqsubseteq$), which exactly corresponds to the part of the source program's behaviour that should be protected by attestation.

Formally, we need to account for the fact that the actual I/O sequence of the remote program includes more information than that of R, e.g., to allow for the cryptographic enforcement of security guarantees. Our definition is parametrised by a $\mathsf{Translate}$ algorithm that permits formalising this notion of *semantic consistency*. Another way to see $\mathsf{Translate}(\mathsf{prms}, \mathsf{ATrace}_{\mathcal{M}_R}(\mathsf{hdl}^*))$ is as a trace translation procedure associated with a given AC scheme, which maps remote traces into traces at the source level.

Definition 9. [Security] An attested computation scheme is secure if there exists an efficient deterministic algorithm $\mathsf{Translate}$ s.t., for all ppt adversaries $\mathcal{A}$, the probability that experiment in Figure 5 (bottom) returns T is negligible.

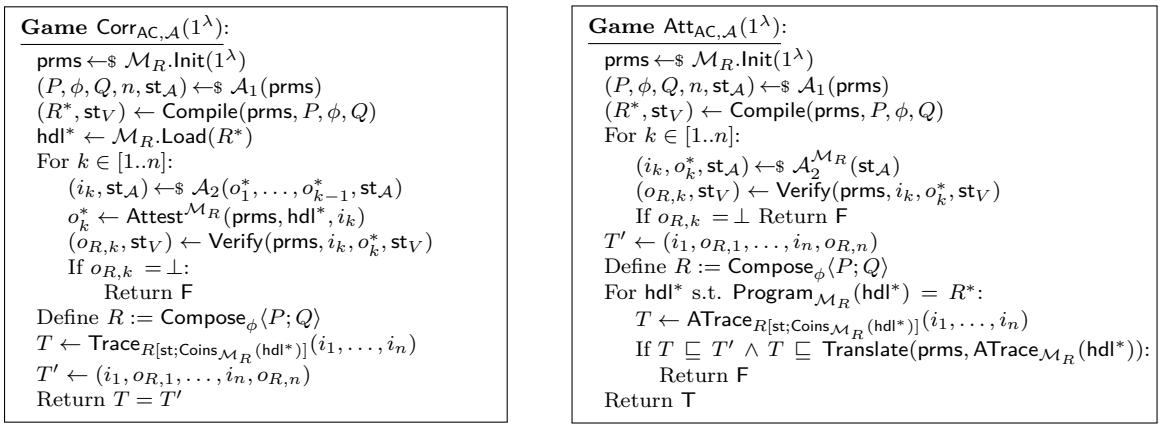

Figure 5: Games defining the correctness (left) and security (right) of an AC scheme.

We note that the adversary loses the game as long as 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 essentially 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 will may easy for the adversary to make the (idealized) local trace diverge from the remote. This is a consequence of our modelling approach, but in no way does it limit the applicability of the primitive we are proposing: it just makes it explicit that the transformation that is 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.

Minimum leakage. From the discussion above, an AC scheme should guarantee 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. However, it is important to establish an additional restriction on what AC compilation actually does to a source program, to ensure that we are able to take advantage of this primitive to achieve more ambitious goals, namely to perform attestation of the remote execution of cryptographic code.

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 when it is remotely executed.

Definition 10. [Minimal leakage]Attested Computation scheme AC ensures security with minimal leakage if it is secure according to Definition 9 and there exists a ppt simulator $\mathcal{S}$ that, for every adversary $\mathcal{A}$, the following distributions are identical:

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

where games $\mathsf{Leak\text{-}Real}_{\mathsf{AC},\mathcal{A}}$ and $\mathsf{Leak\text{-}Ideal}_{\mathsf{AC},\mathcal{A},\mathcal{S}}$ are shown in Figure 6.

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

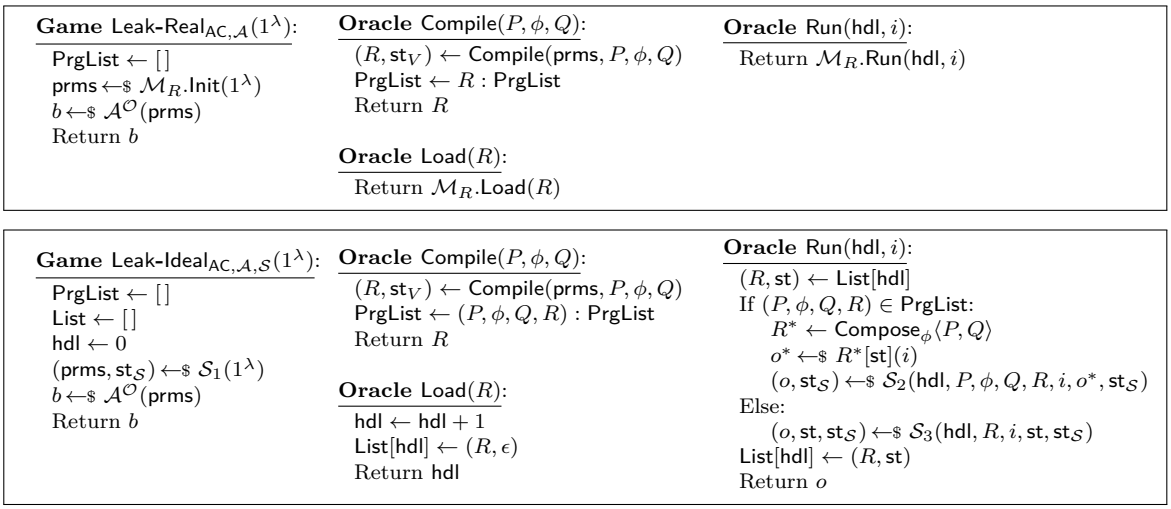

Figure 6: Games defining minimum leakage of an AC scheme.

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).

6 Attested Computation à la SGX

The remote attestation protocol we will consider is inspired in the Secure Guard Extensions (SGX) architecture proposed by Intel [1]. The main feature of this system is that the remote machine is equipped with a security module that manages both short-term and long-term cryptographic keys, with which it is capable of producing MACs that enable authenticated communication between various IEEs and digital signatures that can be publicly verified by anyone holding the (long-term) public key for that machine (or group of machines). We first formalise the operation of (a simplified version of) this security module.

Security module. The security module relies on a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ and a MAC scheme $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Ver})$, and it operates as follows:

- When the host machine is initialised, the security module generates a key pair $(\mathsf{pk}, \mathsf{sk})$ using Σ.Gen and a symmetric key key using Π.Gen. It also creates a special process running code S^* (see below for a description of S^*) in an IEE with handle 0. The security module then securely stores the key material for future use, and outputs the public key. In this case we will have that the output of $\mathcal{M}$.Init will be prms $=$ pk.

- The operation of IEE with handle 0 will be 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 each activation, it expects an input (m, t). It obtains key from the security module and verifies the tag using Π.Ver$(\mathsf{key}, \mathsf{t}, \mathsf{m})$. If the previous oper-

ation was successful, it obtains sk from the security module, signs the message using $\sigma \leftarrow\!\!\$\ \Sigma.\mathsf{Sign}(\mathsf{sk}, \mathsf{m})$ and writes σ to the output. Otherwise, it writes $\perp$ in the output.

- 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).

We note that the operation of the security module allows any process to produce an authenticated message that can be validated by the special process running S^* as coming from within another IEE in the same machine.

We will assume that the message authentication code scheme Π and the signature scheme Σ satisfy the standard notions of correctness and existential unforgeability, and that the machine's public key is authenticated by some external PKI.

Attested Computation scheme. We now define an AC scheme that relies on a remote machine supporting a security module with the above functionality. The operation of the various algorithms is intuitive, except for the fact that basic replay protection using a sequence number does not suffice to bind a remote process to a full trace, since the adversary could then run multiple copies of the same process and *mix and match* outputs from various traces. Instead, the remote process must commit to its entire trace whenever an attested output is produced. Details follow:

- $\mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$ will generate a new program $R^* = \mathsf{Compose}_\phi\langle P^*, Q\rangle$ and output it along with the initial state of the verification algorithm $(R^*, [\,], 1)$, where 1 is an indicator of the stage in which remote program R^* is supposed to be executing. Program P^* is instrumented as follows: it keeps a list ios of all the I/O pairs it has previously received and computed, i.e, its own trace; on each activation with input i, P^* first computes $o \leftarrow\!\!\$\ P[\mathsf{st}_P](i)$ and updates the list by adding a new (i, o) pair; it then requests from the security module a MAC of the updated ios. Due to the operation of the security module, this will correspond to a tag t on the tuple (R^*, ios); it finally outputs $(o, \mathsf{t}, R^*, \mathsf{ios})$. We note that we include (R^*, ios) explicitly in the outputs of R^* for clarity of presentation only. This value would be kept in an insecure environment by a stateful Attest program.

- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i)$ invokes $\mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i)$ using the handle and input value it has received. When the process produces an output o, Attest parses it into $(o', \mathsf{t}, R^*, \mathsf{ios})$. It may happen that parsing fails, e.g., if Q is already executing, in which case Attest simply produces o as its own output. Otherwise, it uses $\mathcal{M}_R.\mathsf{Run}(0, (R^*, \mathsf{ios}, \mathsf{t}))$ to convert the tag into a signature σ on the same message. If this conversion fails, then Attest produces the original output o as its own output. Otherwise, it outputs (o', σ).

- $\mathsf{Verify}(\mathsf{prms}, i, o^*, (R^*, \mathsf{ios}, \mathsf{stage}))$ returns o^* if $\mathsf{stage} = 2$. Otherwise, it first parses o^* into (o, σ), appends (i, o) to ios, and verifies the digital signature σ using prms and $(R^*, \mathsf{ios}))$. If parsing or verification fails, Verify outputs $\perp$. If not, then Verify will check if output o indicates that program P^* has finished. If so, it will update stage to value 2. In any case, it terminates outputting o.

Correctness. It is easy to see that our AC scheme is correct, provided that the underlying signature and message authentication code schemes are themselves correct. To see this, first note that, during the execution of P^*, unless a MAC or signature verification fails, the I/O

sequence produced by Verify exactly matches that of $\mathsf{Compose}_{\phi}\langle P; Q\rangle$, and therefore $T = T'$ is always T. This follows from the construction of R^*, the operation of Attest, and the fact that the associated randomness tapes are established by $\mathsf{Coins}_{\mathcal{M}_R}(\mathsf{hdl}^*)$ as identical. Furthermore, 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 because the combined actions of R^*, Attest, the signing process running S^* and the security module lead to tags and signatures on tuples (R^*, ios) that exactly match those input to the verification algorithms $\Pi.\mathsf{Ver}$ and $\Sigma.\mathsf{Verify}$. Finally, after executing P^*, given that the associated randomness tapes are established by $\mathsf{Coins}_{\mathcal{M}_R}(\mathsf{hdl}^*)$ are identical and that traces are identical up to that point, so will be $\phi(\mathsf{st}_P)$ in both sides, and all subsequent calls to Q will display a similar behaviour.

Security.Let Translate be the deterministic function that receives the machine parameters and a list of tuples of the form $(i, (o, \mathsf{t}, R^*, \mathsf{ios}))$ and returns a list of pairs of the form (i, o).

Theorem 1. *The AC 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 proof of the following theorem can be found in Appendix A. The intuition behind the proof of secure attestation is straightforward: since all attested outputs (i.e. those processed by Verify until $\mathsf{st}_P.\mathsf{finished} = \mathsf{T}$) are bound to a full trace of the execution, all accepted messages that pass AC.Verify must terminate a prefix of a remote trace for some instance of R^*. The only case in which the adversary could win would be if the signature verification performed by Verify accepts a message that was never authenticated by an IEE running R^*. However, in this case, the adversary is either breaking the MAC (to dishonestly execute Attest) or breaking the signature (and forging attested outputs directly). The minimum leakage property can be proven by constructing the trivial simulator that generates the machine parameters itself, simulates the entire machine for non-attested processes, and attaches MACs to the source I/O traces of attested programs.

7 AKE for Attested Computation

An intermediate step in constructing high-level applications that rely on attested computation is the establishment of a secure communications channel with a process running a particular program inside an IEE in the remote machine. After such a channel has been established, standard cryptographic techniques can be used to ensure (in combination with the isolation provided by IEEs) the integrity and confidentiality of subsequent computations. In this section we will see how attested computation, in combination with a specific flavour of key exchange protocol can be seen as a bootstrapping process for this scenario.

We first formalize the precise requirements for a key exchange protocol that can be used in this setting (we call this *authenticated key exchange for attested computation*) and show how a simple transformation can be used to construct such protocols from any passively secure key exchange protocol. Later on we present a utility theorem that precisely describes what it means to use attested computation and a suitable key exchange protocol to establish a secure channel with an arbitrary remote program.

Definitions

SYNTAX. A *Key Exchange for Attested Computation* (AttKE) protocol is defined by the following pair of algorithms.

- $\mathsf{Setup}(1^\lambda, \mathsf{id})$ is the remote program generation algorithm, which is run on the local machine to initialise a fresh instance of the AttKE protocol under party identifier id. On input the security parameter and id, it will output the code for a program $\mathsf{Rem}_{\mathsf{KE}}$ and the initial state st_L of the $\mathsf{Loc}_{\mathsf{KE}}$ algorithm. This algorithm is run locally.
- $\mathsf{Rem}_{\mathsf{KE}}$ (which is generated dynamically by Setup) is a program that will be run as a part of an IEE process in the remote machine, and it will keep the entire remote state of the key exchange protocol in that protected environment.
- $\mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_L, \mathsf{m})$ is the algorithm that runs the local end of the AttKE protocol, interacting with $\mathsf{Rem}_{\mathsf{KE}}$. On input its current state, and an incoming message m, it will output an updated state and an outgoing message.

When analysing the security of such a protocol we will impose that the $\mathsf{Loc}_{\mathsf{KE}}$ algorithm and all $\mathsf{Rem}_{\mathsf{KE}}$ programs that may be produced by Setup keep in their state the same information that was imposed on general key exchange algorithms in Section 3.3. We will refer to the instances of local key exchange executions as $\mathsf{Loc}^s_{\mathsf{KE}}$, for $s \in \mathbb{N}$. The local identity will be implicit in our notation since, in the following discussion we will concentrate our attention on the case where a single local identity id is considered. We do this for the sake of rigour and clarity of presentation: by looking at this simplified case we can present our security models in game-based form, whilst taming the complexity of the resulting games. The extension of these results to the more general case where several local identities are considered is straightforward. On the remote side, the identity of the remote process will actually be generated on the fly by the combined actions of the Setup algorithm and possibly the protocol execution itself, as it may depend for example on the code of the remote program. For this reason we will enumerate over remote instances as $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$ for $i, j \in \mathbb{N}$, and observe that the value of variable oid in this case will be set during the execution of the program itself, rather than passed explicit as an input to one of the algorithms. CORRECTNESS. An AttKE is correct if, after a complete (honest) run between two participants, one local and one remote, and where the remote program is always the one to initiate the communication, both reach the accept state, both derive the same key and session identifier and have matching partner identities. More formally, a protocol $P = \{\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}}\}$ is correct if, for any arbitrary identity id, the experiment in Figure 7 always returns T. We note that our definition of correctness imposes that remote programs always operate as initiators and local machines as the responders in the key exchange.

EXECUTION ENVIRONMENT. The specific flavour of key exchange that we will be considering is clarified by the execution environment in Figure 8. This follows the standard modelling of active attackers, e.g. [20], when one excludes the possibility of corruption (which we do only for the sake of simplicity). There are, however, two modifications that attend to the fact that AttKE remote programs are designed to be executed under attested computation guarantees. On one hand, the adversary is given the power to create as many remote AttKE programs as it may need, by using the $\mathsf{NewLocal}$ oracle. This reveals the entire code of the remote AttKE program (and implicitly all of its initial internal state, which is assumed to be

Game $\mathsf{Corr}_{\mathsf{AttKE}}(1^\lambda)$:

$(\mathsf{st}_L, \mathsf{Rem}_{\mathsf{KE}}) \leftarrow\$\ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$\mathsf{st}_R \leftarrow \epsilon$
$\mathsf{m} \leftarrow\$\ \mathsf{Rem}_{\mathsf{KE}}[\mathsf{st}_R](\epsilon)$
$\mathsf{t} \leftarrow \mathsf{T}$
While $\mathsf{m} \neq \epsilon$:
  If t: $(\mathsf{st}_L, \mathsf{m}) \leftarrow\$\ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_L, \mathsf{m})$
  Else: $\mathsf{m} \leftarrow\$\ \mathsf{Rem}_{\mathsf{KE}}[\mathsf{st}_R](\mathsf{m})$
  $\mathsf{t} \leftarrow \neg\mathsf{t}$
Return $\mathsf{st}_L.\delta = \mathsf{st}_R.\delta = \mathsf{accept} \wedge \mathsf{st}_L.\mathsf{key} = \mathsf{st}_R.\mathsf{key} \wedge \mathsf{st}_L.\mathsf{sid} = \mathsf{st}_R.\mathsf{sid} \wedge$
  $\mathsf{st}_L.\mathsf{pid} = \mathsf{st}_R.\mathsf{oid} \wedge \mathsf{st}_L.\mathsf{oid} = \mathsf{st}_R.\mathsf{pid} \wedge \mathsf{st}_L.\mathsf{oid} = \mathsf{id}$

Figure 7: Game defining the correctness of an AttKE scheme.

empty)to the adversary. This captures the fact that remote AttKE programs will be loaded into IEE execution environments in an otherwise untrusted remote machine, and it implies that remote AttKE programs cannot keep *any* long term secret information. Intuitively, this limitation will be compensated by the attested computation protocol. On the other hand, the adversary is able to freely interact with remote processes, but it is constrained in its interaction with the local machine. Indeed, the SendLocal oracle filters which messages the adversary can deliver to the local machine by checking that these are consistent with at least one remote process that the adversary is interacting with. This captures the fact that AttKE is designed to interact over a partially authenticated channel from the remote machine to the local machine, which will be provided by an attested computation protocol.

Game $\mathsf{Att}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda)$:

$\mathsf{InsList} \leftarrow [\,]$; $\mathsf{fake} \leftarrow [\,]$
$i \leftarrow 0$
$b \leftarrow\$\ \{0,1\}$
$b' \leftarrow\$\ \mathcal{A}^{\mathcal{O}}(1^\lambda, \mathsf{id})$
Return $b = b'$

Oracle NewLoc():

$i \leftarrow i+1$; $T_L^i \leftarrow []$
$(\mathsf{Rem}_{\mathsf{KE}}^i, \mathsf{st}_L^i) \leftarrow\$\ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$\mathsf{InsList}[i] \leftarrow 0$
Return $\mathsf{Rem}_{\mathsf{KE}}^i$

Oracle TestLoc(i):

If $\mathsf{st}_{\mathsf{L}}^i.\delta \neq \mathsf{accept}$ return $\perp$
If $b = 0$ return $\mathsf{st}_L^i.\mathsf{key}$
Return $\mathsf{fake}(\mathsf{st}_L^i.\mathsf{key})$

Oracle SendLoc(m, i):

If $\nexists j$, $(\mathsf{m} : T_L^i) \sqsubseteq T_R^{i,j}$ return $\perp$
$(\mathsf{m}', \mathsf{st}_L^i) \leftarrow\$\ \mathsf{Loc}_{\mathsf{KE}}^i(\mathsf{st}_L^i, \mathsf{m})$
$T_L^i \leftarrow \mathsf{m}' : \mathsf{m} : T_L^i$
If $\mathsf{st}_L^i.\delta \in \{\mathsf{accept}, \mathsf{derived}\}$:
  If $(\mathsf{st}_L^i.\mathsf{key}, \mathsf{key}^*) \notin \mathsf{fake}$:
    $\mathsf{key}^* \leftarrow\$\ \{0,1\}^\lambda$
    $\mathsf{fake} \leftarrow (\mathsf{st}_L^i.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return $(\mathsf{m}', \mathsf{st}_L^i.\mathsf{sid}, \mathsf{st}_L^i.\delta, \mathsf{st}_L^i.\mathsf{pid})$

Oracle RevealLoc(i):

Return $\mathsf{st}_L^i.\mathsf{key}$

Oracle RevealRem(i, j):

Return $\mathsf{st}_R^{i,j}.\mathsf{key}$

Oracle NewRem(i):

$\mathsf{InsList}[i] \leftarrow \mathsf{InsList}[i] + 1$
$j \leftarrow \mathsf{InsList}[i]$
$T_R^{i,j} \leftarrow [\,]$; $\mathsf{st}_R^{i,j} \leftarrow \epsilon$
Return ϵ

Oracle TestRem(i, j):

If $\mathsf{st}_{\mathsf{R}}^{i,j}.\delta \neq \mathsf{accept}$ return $\perp$
If $b = 0$ return $\mathsf{st}_R^{i,j}.\mathsf{key}$
Return $\mathsf{fake}(\mathsf{st}_R^{i,j}.\mathsf{key})$

Oracle SendRem(m, i, j):

// No restriction
$\mathsf{m}' \leftarrow\$\ \mathsf{Rem}_{\mathsf{KE}}[\mathsf{st}_R^{i,j}](\mathsf{m})$
$T_R^{i,j} \leftarrow \mathsf{m}' : \mathsf{m} : T_R^{i,j}$
If $\mathsf{st}_R^{i,j}.\delta \in \{\mathsf{accept}, \mathsf{derived}\}$:
  If $(\mathsf{st}_R^{i,j}.\mathsf{key}, \mathsf{key}^*) \notin \mathsf{fake}$:
    $\mathsf{key}^* \leftarrow\$\ \{0,1\}^\lambda$
    $\mathsf{fake} \leftarrow (\mathsf{st}_R^{i,j}.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return $(\mathsf{m}', \mathsf{st}_R^{i,j}.\mathsf{sid}, \mathsf{st}_R^{i,j}.\delta, \mathsf{st}_R^{i,j}.\mathsf{pid})$

Figure 8: Execution environment for AttKEs.

Partnering. We will consider the natural extension of the partnering properties introduced for passive key exchange in Section 3.3to the AttKE setting. In addition to the

syntactic modifications that result from referring to $\mathsf{Loc}^s_{\mathsf{KE}}$ and $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$, we further restrict validity so that partnering is only valid when it occurs between local and remote instances, in which the latter is the initiator. To this end, we will use the following predicate on two instances $\mathsf{Loc}^s_{\mathsf{KE}}$ and $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$ holding $\mathsf{st}^s_L = (\mathsf{st}^s, \delta^s, \rho^s, \mathsf{sid}^s, \mathsf{pid}^s, \mathsf{oid}^s, \mathsf{key}^s)$ and $\mathsf{st}^{i,j}_R = (\mathsf{st}^{i,j}, \delta^{i,j}, \rho^{i,j}, \mathsf{sid}^{i,j}, \mathsf{pid}^{i,j}, \mathsf{oid}^{i,j}, \mathsf{key}^{i,j})$, respectively:

$$\mathrm{P}(\mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \begin{cases} \mathsf{T} \text{ if } \mathsf{sid}^s = \mathsf{sid}^{i,j} \ \wedge \ \delta^s, \delta^{i,j} \in \{\mathsf{derived}, \mathsf{accept}\} \\ \mathsf{F} \text{ otherwise.} \end{cases}$$

The definition of partner is the obvious one, whereas invalid partners now includes an extra possibility.

Definition 11 (Partner)**.** Two instances $\mathsf{Loc}^s_{\mathsf{KE}}$ and $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$ are partnered if

$$\mathrm{P}(\mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T}\,.$$

Definition 12 (Valid Partners)**.** A protocol AttKE ensures valid partners if the bad event notval does not occur, where notval is defined as one of the following events occurring:

$$\begin{aligned} \exists \mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}} \text{ s.t. } \mathrm{P}(\mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T} \wedge (\, \mathsf{pid}^s \neq \mathsf{oid}^{i,j} \ \vee \ \mathsf{oid}^s \neq \mathsf{pid}^{i,j} \vee \\ \mathsf{key}^s \neq \mathsf{key}^{i,j} \ \vee \ \rho^s \neq \mathsf{responder} \ \vee \ \rho^{i,j} \neq \mathsf{initiator}\,) \\ \exists \mathsf{Loc}^r_{\mathsf{KE}}, \mathsf{Loc}^s_{\mathsf{KE}} \text{ s.t. } r \neq s \ \wedge \ \mathrm{P}(\mathsf{Loc}^r_{\mathsf{KE}}, \mathsf{Loc}^s_{\mathsf{KE}}) = \mathsf{T} \\ \exists \mathsf{Rem}^{i,j}_{\mathsf{KE}}, \mathsf{Rem}^{k,l}_{\mathsf{KE}} \text{ s.t. } (i,j) \neq (k,l) \ \wedge \ \mathrm{P}(\mathsf{Rem}^{i,j}_{\mathsf{KE}}, \mathsf{Rem}^{k,l}_{\mathsf{KE}}) \end{aligned}$$

For completeness, we present also the adapted definitions of confirmed and unique partners.

Definition 13 (Confirmed Partners)**.** A protocol AttKE ensures confirmed partners if the bad event notconf does not occur, where notconf is defined as at least one of the following two events occurring:

$$\begin{aligned} \exists \mathsf{Loc}^s_{\mathsf{KE}} \text{ s.t. } \delta^s = \mathsf{accept} \quad \wedge \quad \forall\, \mathsf{Rem}^{i,j}_{\mathsf{KE}},\ \mathrm{P}(\mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{F} \\ \exists \mathsf{Rem}^{i,j}_{\mathsf{KE}} \text{ s.t. } \delta^{i,j} = \mathsf{accept} \quad \wedge \quad \forall\, \mathsf{Loc}^s_{\mathsf{KE}},\ \mathrm{P}(\mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{F}. \end{aligned}$$

Definition 14 (Unique Partners)**.** A protocol AttKE ensures unique partners if the bad event notuni does not occur, where notuni is defined as at least one of the following two events occurring:

$$\begin{aligned} \exists \mathsf{Loc}^s_{\mathsf{KE}}, \quad & \mathsf{Rem}^{i,j}_{\mathsf{KE}}, \mathsf{Rem}^{i',j'}_{\mathsf{KE}} \text{ s.t.} \\ & (i,j) \neq (i',j') \wedge \mathrm{P}(\mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T} \wedge \mathrm{P}(\mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i',j'}_{\mathsf{KE}}) = \mathsf{T} \\ \exists \mathsf{Rem}^{i,j}_{\mathsf{KE}}, \quad & \mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Loc}^{s'}_{\mathsf{KE}} \text{ s.t.} \\ & s \neq s' \wedge \mathrm{P}(\mathsf{Loc}^s_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T} \wedge \mathrm{P}(\mathsf{Loc}^{s'}_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T}\,. \end{aligned}$$

As before, we will consider that an adversary violates entity authentication if he can get a session to accept, but there is no unique and confirmed valid session in its intended partner.More formally, we wish to verify that none of the bad events notval, notconf, notuni occurs. In the attested computation scenario, it is common to use one-sided authentication where only the local party receives authentication guarantee. Such definitions can be easily derived from the ones we have presented above, analogously to what was done in Section 3.3.

Security. Again, the set of TestLoc and TestRem queries must be restricted in order to exclude trivial attacks. An adversary is legitimate if it respects the following freshness criteria:

- For all $\mathsf{TestLoc}(i)$ queries, the following holds:
  1. $\mathsf{RevealLoc}(i)$ was not queried; and
  2. for all $\mathsf{Rem}_{\mathsf{KE}}^{j,k}$ s.t. $\mathrm{P}(\mathsf{Rem}_{\mathsf{KE}}^{j,k}, \mathsf{Loc}_{\mathsf{KE}}^{s}) = \mathsf{T}$, $\mathsf{RevealRem}(j,k)$ was not queried.
- For all $\mathsf{TestRem}(i,j)$ queries, the following holds:
  1. $\mathsf{RevealRem}(i,j)$ was not queried; and
  2. for all $\mathsf{Loc}_{\mathsf{KE}}^{k}$ s.t. $\mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^{k}, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{T}$, $\mathsf{RevealLoc}(i)$ was not queried.

We only consider legitimate adversaries, and say that the winning event guess occurs if $b = b'$ at the end of the experiment. We define AttKE security by requiring both mutual authentication of parties and and key secrecy.

Definition 15 (AttKE security)**.** An AttKE protocol is secure if, for any ppt adversary in Figure 8, and for any local party identifier string id: 1. the adversary violates entity authentication with negligible probability $\Pr[\,\mathsf{notval} \vee \mathsf{notconf} \vee \mathsf{notuni}\,]$; and 2. its key secrecy advantage $2 \cdot \Pr[\,\mathsf{guess}\,] - 1$ is negligible.

Generic Construction

We now present a construction of an AttKE scheme from any passively secure key exchange protocol, relying additionally on a existentially unforgeable signature scheme. The intuition here is that the attested computation protocol guarantees correct remote execution of a program, but does not ensure uniqueness, i.e., it does not exclude that potentially many replicas of the same key exchange protocol instance could be running in the remote machine. By binding a fresh signature verification key with the identifier for the remote party associated with the key exchange protocol and generating a fresh nonce at the start of every execution, we can remotely execute the key exchange code whilst ensuring one-to-one authentication at the process level. This transformation can be seen as a weaker version of the well-known passive-to-active compilation process by Katz et al. [20], since our target security model is not fully active. We now present the details.

Consider a passively-secure authenticated key exchange protocol Π and a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$. Our construction splits the execution of Π between the local machine and a remote isolated execution environment: the responder will run locally and the initiator will run remotely within a program $\mathsf{Rem}_{\mathsf{KE}}$.[4] The code of the remote program will have hardwired into it a unique verification key for the signature scheme. The first activation of $\mathsf{Rem}_{\mathsf{KE}}$ initialises an internal state and computes a nonce, together with the first message in the key exchange protocol. The party identifier string of the remote process will then be defined to comprise the verification key and the nonce. The local part of the protocol signs the full communication trace so far. Subsequent activations of remote program $\mathsf{Rem}_{\mathsf{KE}}$ will simply respond according to the key exchange protocol description, rejecting all inputs that fail signature verification. The details of our construction are shown in Figure 9.

- Setup first generates a fresh key pair for the signature scheme and constructs program $\mathsf{Rem}_{\mathsf{KE}}$, parametrised by algorithm Π and verification key pk, as described in Figure 9 (top). In this program state variables δ, ρ, key, sid and pid are all shared with Π (this

[4]Setting the remote machine as the initiator of the protocol is the most common scenario. We considered it for simplicity; the converse can be treated analogously.

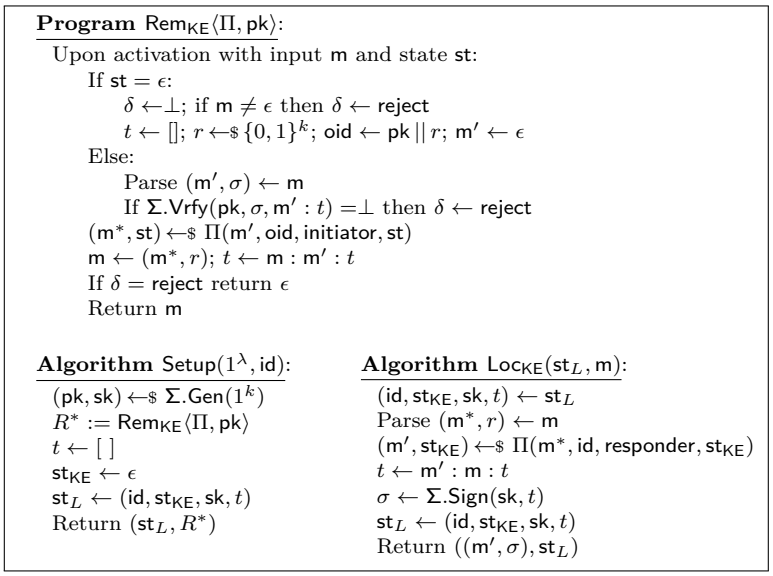

Figure 9: Details of the AttKE construction.

is implicit in the figure). The initial value of st_L will store id, along with the initially empty state for the key exchange $\mathsf{st}_K E$, the signing key for the signature scheme and an initially empty trace t log.

- $\mathsf{Loc}_{\mathsf{KE}}$ takes $(\mathsf{st}_L, \mathsf{m})$ runs $\Pi(\mathsf{m}, \mathsf{id}, \mathsf{responder}, \mathsf{st}_{\mathsf{KE}})$ to compute the next message o, produces signature σ of the entire updated protocol trace, and returns the updated state st_L and message (o, σ).

The following theorem establishes the correctness and security of the generic construction.

Theorem 2. *Given a correct passively secure key exchange protocol* Π *and an existentially unforgeable signature scheme* Σ*, the generic construction above yields a correct and secure AttKE protocol.*

The full proof is given in Appendix BThe intuition behind the proof is that each local instance of the key exchange protocol is bound to a verification key for the signature scheme which is hardwired into the party identifier of the associated remote program code. Each remote instance of the code initialises its own party identifier by attaching a nonce to the verification key. This nonce is also transmitted along with every remote-to-local message. These facts combined with the restriction on the $\mathsf{SendLoc}$ oracle imply that the adversary is essentially restricted to function in a passive way, by passing around messages between the two Send oracles. Therefore the security guarantees can be reduced to the security of the underlying key exchange protocol.

Utility

As an intermediate result building up to the construction of full-fledged authenticated and private remote attested computation, we will now present a utility theorem that describes precisely the guarantees one obtains when combining an attested computation protocol with an AttKE. Intuitively, this theorem states that attested computation guarantees that the authentication and secrecy assurance offered by AttKE are retained when we use it to establish

session keys with remote IEEs, in the presence of fully active adversaries that control the remote machine, and when the key exchange is composed with arbitrary programs.

Figure 10 shows an idealised game where an adversary must distinguish between two remote machines where an AttKE scheme is executed in combination with an AC scheme. Machine $\mathcal{M}_R$ is any standard remote machine that is supported by the attested computation protocol, whereas $\mathcal{M}'_R$ represents a modification of $\mathcal{M}_R$ where one can tweak the operation of $\mathsf{Rem}_{\mathsf{KE}}$ programs. The differences of $\mathcal{M}'_R$ with respect to $\mathcal{M}_R$ are concentrated on the Run interface, which now operates as follows:

- It takes as additional parameters a list fake of pairs of keys and Boolean flag tweak that, when activated, identifies a process that is running an instance of $\mathsf{Rem}_{\mathsf{KE}}$ composed with some program Q. This flag triggers the following modifications with respect to the operations of $\mathcal{M}_R$.
- When it detects that $\mathsf{Rem}_{\mathsf{KE}}$ has transitioned into derived or accept state, it will check if the derived key exists in list fake. If not, it generates a new random key^*, and $(\mathsf{key}, \mathsf{key}^*)$ is added to the list.
- When it detects that program Q is set to start executing, rather than using the key as an input to ϕ, it uses $\mathsf{fake}(\mathsf{key})$ instead.

The environment presented to the adversary models a standard attested computation interaction, where it is given total control over the remote machine using oracles Load and Run (these oracles will either give access to $\mathcal{M}_R$ or to $\mathcal{M}'_R$, depending on a secret bit b generated in the beginning of the game). The adversary is also able to obtain challenge remote programs using a $\mathsf{NewSession}(Q)$ oracle that uses the attested computation scheme to compile $\mathsf{Rem}_{\mathsf{KE}}$ composed with arbitrary program Q of its choice under a mapping function ϕ_{key} that reveals the relevant parts of the key exchange state (namely the secret key key, the party identifiers oid and pid, the state δ and the session identifier sid). We observe that such arbitrary programs can leak all of the information revealed by ϕ_{key} to the attacker. If the adversary chooses to Load a challenge program, and if $\mathcal{M}'_R$ is being used in the game, then it will be tweaked as described above. Whenever $\mathsf{NewSession}(Q)$ is called, the environment creates a new local session i that the adversary can interact with using a $\mathsf{Send}(i, \mathsf{m})$ oracle. The Send oracle uses the Verify algorithm of the attested computation scheme to validate attested outputs and, if they are accepted, feeds them to the $\mathsf{Loc}_{\mathsf{KE}}$ instance (and also ensures that list fake is updated). Finally, the adversary can explicitly choose to be tested (as opposed to the implicit testing it may trigger using arbitrary programs Q) by calling Test on a local instance. This oracle will either return the true key, if $b = 0$, or the associated random key that is kept in the fake list. As before, we define the winning event guess to occur when $b = b'$ in the end of the game.

The proof of the following theorem can be found in Appendix C.

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

The intuition behind the proof is that one can use the security of attested computation to exclude Send queries that do not match a legitimate remote trace. This essentially maps

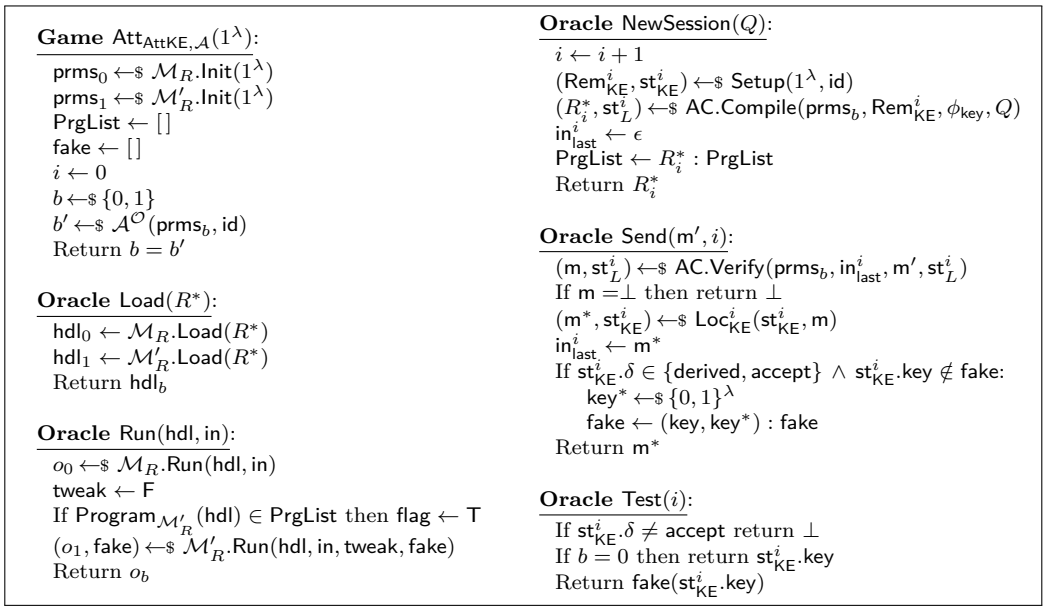

Figure 10: Game defining the utility of an AttKE scheme when used in the context of attested computation.

to the restriction in the AttKE security game imposed on the SendLoc oracle. The proof is concluded by applying the minimum leakage property of the attested computation protocol to show that any attack by the adversary against AttKE when it is run inside a remote machine can be transformed (via trace simulation) to an attack against the original AttKE when it is run in source code form.

8 Secure Outsourced Computation

In this section we build on the results in previous sections to design and analyze a protocol for secure outsourced computation. Informally, we require two properties i) that only the legitimate local user can pass inputs to the outsourced program and ii) that the I/O of the remote program is secret from any observer (even an actively malicious one).

We first give syntax for the protocols that solve this problem, then propose formal definitions for the properties that we outlined above and conclude with a generic construction that combines a key-exchange for attestation, a scheme for attested computation and an authenticated encryption scheme.

Syntax. A *Secure Outsourced Computation* scheme (SOC) for a remote machine $\mathcal{M}_R$ is defined by the following algorithms:

- $\mathsf{Compile}(\mathsf{prms}, P, \mathsf{id})$ is the program compilation algorithm. On input a program P and a party identifier id, it outputs a compiled program P^*, together with an initial state st_l for the local side algorithms. We assume that initially $\mathsf{st}_l.\mathsf{accept} = \perp$. Note that unlike the AC compilation algorithm, this algorithm only takes one program as input, as this scheme is intended for providing guarantees for the whole trace and not only an initial segment.
- $\mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$ is the client side initialization algorithm. On input o (presumably

the last message from the remote machine) it returns the next message i to be delivered to the remote machine in the bootstrapping step step, together with the updated local state. We assume $\mathsf{BootStrap}$ sets an accept flag to T when the initialization process successfully terminates.

- $\mathsf{Verify}(\mathsf{prms}, o^*, \mathsf{st}_l)$ is the verification algorithm. It fulfills the same function as the AC verification algorithm. Note that, as all the inputs are provided by the local machine, we do not need to feed it the last input as it can be stored in the state. It is expected to return $\perp$ if $\mathsf{st}_l.\mathsf{accept} \neq \mathsf{T}$.

- $\mathsf{Encode}(\mathsf{prms}, i, \mathsf{st}_l)$ is the encoding algorithm. On input the local state and the next intended input for P, it returns the next input i^* for P^* together with the updated local state. It is expected to return $\perp$ if $\mathsf{st}_l.\mathsf{accept} \neq \mathsf{T}$.

- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i)$ is, as in an AC scheme, the (untrusted) attestation algorithm.

A party A with identifier id who wants to outsource program P to the remote machine first compiles P with his id, thus obtaining P^* and some secret data st_l. He then loads P^* on the remote machine using some untrusted protocol. As it is, the program P^* is not ready to receive inputs intended for P: an initial bootstrapping phase (until $\mathsf{BootStrap}$ sets the accept flag) is necessary to establish some shared secrets between the IEE in which P^* is executed and A. Then when A wants to send an input to the remote execution, he encodes it using Encode, sends it (using Attest) and verifies the output provided by Attest using Verify.

In this section, for simplicity reasons, we assume that the program P is deterministic. However, as for an AC scheme it would be easy to extend all the definitions to a non-deterministic program.

Input Integrity. While security of attested computation aims at ensuring that a trace was honestly produced on the remote side, it does nothing to restrict the provenance of the inputs received.

We provide a stronger notion which we call *input integrity* which, intuitively, ensures that if a program is compiled by a party with identifier id, then only that party may use the remote compiled program. We ensure this property by making sure that the local and remote views coincide (up to the last message exchanged, which may not have yet been delivered). The following formula Ψ which relates two input/output traces captures this intuition.

$$\Psi(T, T') := T = T' \ \vee \ \exists o.(T = o :: T') \ \exists i.(T' = i :: T)$$

The formalization that we provide in Figure 11 is as follows. The adversary chooses a program P that is compiled with an honest party's id yielding P^* (which is given to the adversary). The adversary is given access to two oracles. A bootstrapping oracle that simply executes $\mathsf{BootStrap}$ honestly; and a send oracle that verifies the last (presumed) output of the remote program and encodes the next input (which is provided by the adversary), while keeping track of the local view of the trace. The goal of the adversary is then create a mismatch between the local and remote view of the trace.

Definition 16 (Input Integrity)**.** We say that a SOC scheme satisfies *input integrity* if there exists a polynomial time algorithm $\mathsf{Translate}$ such that for all ppt $\mathcal{A}$ the experiment described in Figure 11 returns true with probability negligible in the security parameter.

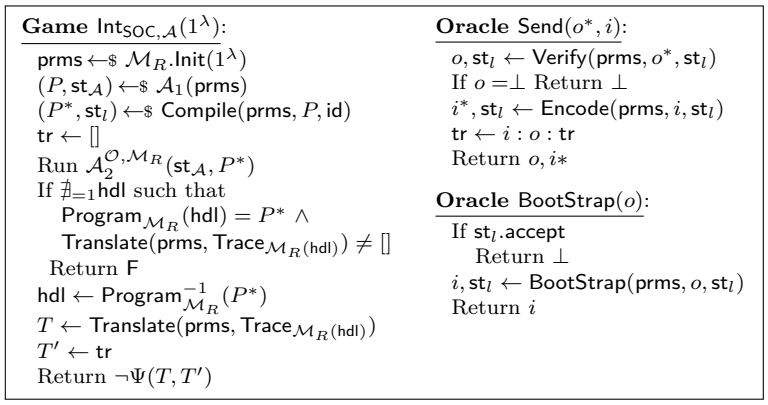

Figure 11: Input integrity of a SOC scheme

Input privacy. We define the privacy of I/O with an indistinguishability game. One important point here is that we chose to restrict the class of programs we consider to *length-uniform* (written lu) programs. A program is length uniform if the length of its outputs depends only on the length of its inputs. Intuitively, this is because the encryption scheme is allowed to leak the length of the messages, which in turn would leak information about the inputs for a non lu program.

The formalization described in Figure 12 is as follows. We start by choosing a bit b that will determine whether the adversary will be talking with the left send oracle or the right send oracle (described later). As for input integrity, the adversary then chooses a program P. We compile it for an honest party's identifier and give the resulting P^* to the adversary. The adversary is also given access to the bootstrapping oracle. In addition, he is given access to a left or right send oracle. This oracle, on a request with the last candidate output of the remote machine and two inputs i_0 and i_1, verifies the last candidate output and, depending on the bit b, encodes either i_0 or i_1 and returns the result. The goal of the adversary is to guess the bit b with non-negligible bias from 1/2.

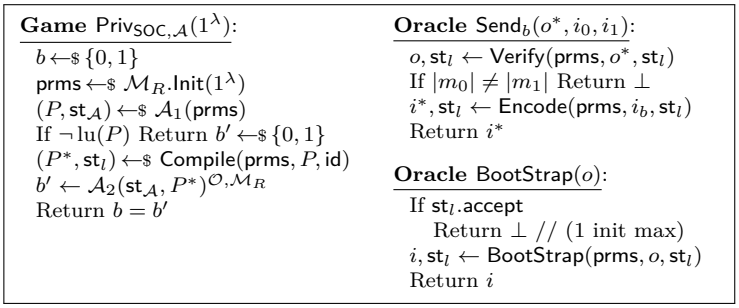

Figure 12: Input privacy of a SOC scheme

Definition 17. We say that a SOC scheme satisfies *input privacy* if, for all ppt $\mathcal{A}$, the experiment in Figure 12 returns true with probability 1/2 up to a negligible function.

This definition ensures that there exist no two traces (with messages of the same length) played by an honest party over a SOC protocol that are distinguishable for an (active) adversary. This means that no adversary can gain information on the inputs sent out by a local machine using a SOC scheme, besides the length of the messages exchanged, achieving our goal of hiding the honest party's inputs.

Definition 18. We say that a SOC scheme is *secure* if it satisfies both input privacy and input integrity.

An implementation of a secure SOC scheme. Having defined what security we expect from a SOC scheme, we now define a scheme that satisfies these requirements. We base our construction on an AttKE, and an AC scheme. The main idea is using the AttKE to establish a key between the party agent and the IEE, and then communicate with the IEE over the secure channel established with this key.

Formally, let $(\mathsf{Compile}, \mathsf{Attest}, \mathsf{Verify})$ be an AC scheme, $(\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}})$ be an AttKE and (E, D, K) be an authenticated encryption scheme. Figure 13 defines a SOC scheme. The most important part is the compilation part, which uses the AC scheme compilation to compile the composition of the $\mathsf{Rem}_{\mathsf{KE}}$ program generated by Setup together with program P running over a secure channel (denoted by $C(P)$). The initial local state is the union of the state provided by the AC compilation and the AttKE setup. The program $C(P)$ simply decrypts the message it receives checks that the sequence number of the message matches its view the passes the decrypted message to P. It then retrieves the output of P, appends the corresponding next sequence number and outputs it. This mechanism ensures that all messages received (resp. sent out) by P^* after the bootstrapping phase have the form $E(i\#m, k)$ where i is the position of the message in the trace, m is the message intended to (resp. produced by) P, and k is the key established by the AttKE.

On the local side, the bootstrapping mechanism simply consists of running the local KE over the AC protocol as already described in the utility definition. Once the key has been established, the local state keeps track of the local view of the sequence number. Verifying an output consists in decrypting it and checking that the sequence number against the local view of it. Encoding an input, is just appending the correct sequence number and encrypting it with the shared key.

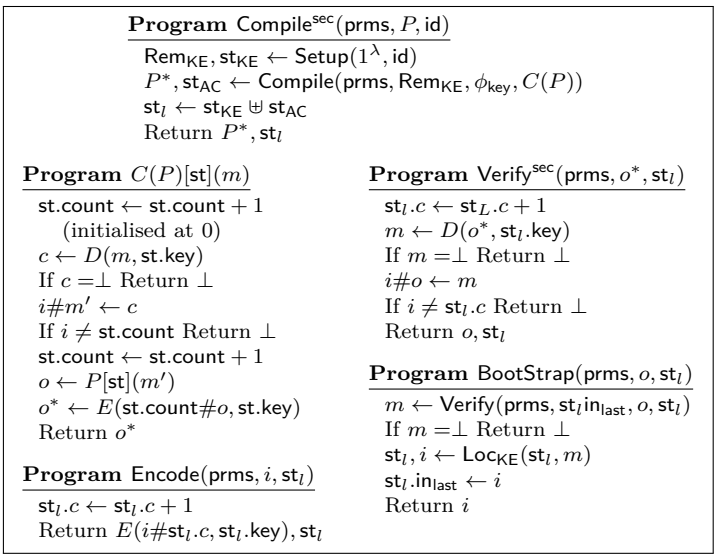

Figure 13: SOC algorithms

Theorem 4. *If* $(\mathsf{Compile}, \mathsf{Attest}, \mathsf{Verify})$ *is a correct and secure AC scheme,* $\{\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}}, \mathsf{Rem}_{\mathsf{KE}}\}$ *is a secure AttKE and* (E, D, KG) *is an secure authenticated encryption scheme, then the SOC presented in Figure 13 is secure.*

The proof is provided in Appendix D, we provide here a sketch of proof. We first do a game hop that consists in replacing the key established by P^* and the party using the AttKE by a "magically" shared fresh key. The utility property of the AttKE provides us with the fact that we can replace the key shared by the remote machine and the local agent by a freshly generated key. We are left with showing that this key is shared with an IEE which is indeed running P^* and not $\mathsf{Compile}(\mathsf{Rem}_{\mathsf{KE}}, \phi_{\mathsf{key}}, Q)$ for some other Q, this is provided by the security of the AC scheme.

We then prove input integrity by remarking that injecting new messages in the trace would contradict the unforgeability of the authenticated encryption scheme. The sequence number ensures that the messages are delivered in the right order and that replays are impossible.

We remark that the input integrity property ensures that we know that the only meaningful action the adversary can take is to forward messages between the remote and local machines. Taking advantage of that fact we can reduce the input privacy game to the IND-CPA property of the authenticated encryption.

9 Conclusion

This paper offers a set of building blocks for constructing protocols that leverage the guarantees of IEEs. First, we define and construct attested computation based on IEEs. In the process we identify and formalize two key properties that such protocols need to satisfy to be useful: composition awareness and minimal leakage. Our instantiation of attested computation relies on SGX.

The next component that we provide are key-exchange protocols between a remote party and an IEE. Such protocols are components which need to be used in any protocol where secrecy of data communicated to and from the IEE is important. Our contribution is to adapt existing models of security for key-exchange to the novel setting where protocol participants do not necessarily have an a-priori identity – IEEs cannot be uniquely identified before actively communicating with them. For constructions, we present a modular approach where by combining a passively secure key exchange protocol with an arbitrary attested computation protocol we obtain a fully secure key-exchange protocol. As an application, we show how to use attested computation, key-exchange and symmetric authenticated encryption to generically construct a secure outsourced computation scheme for arbitrary functionalities.

In terms of follow-up work, the natural next step is to use the building blocks that we provide to construct other protocols. One interesting target (which generalizes the application in this paper) is to construct secure multi-party computation based on IEEs and experimentally compare their efficiency with the existent software-only alternatives.

References

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013.

[2] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems*

Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014., pages 267–283. USENIX Association, 2014.

[3] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358. Springer, 1994.

[4] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 111–131. Springer, 2011.

[5] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 132–145. ACM, 2004.

[6] Ernie Brickell, Liqun Chen, and Jiangtao Li. A new direct anonymous attestation scheme from bilinear maps. In *Trusted Computing - Challenges and Applications, First International Conference on Trusted Computing and Trust in Information Technologies, Trust 2008, Villach, Austria, March 11-12, 2008, Proceedings*, volume 4968 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 2008.

[7] Christina Brzuska, Nigel P. Smart, Bogdan Warinschi, and Gaven J. Watson. An analysis of the EMV channel establishment protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 373–386. ACM, 2013.

[8] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

[9] Luigi Catuogno, Alexandra Dmitrienko, Konrad Eriksson, Dirk Kuhlmann, Gianluca Ramunno, Ahmad-Reza Sadeghi, Steffen Schulz, Matthias Schunter, Marcel Winandy, and Jing Zhan. Trusted virtual domains - design, implementation and lessons learned. In *Trusted Systems, First International Conference, INTRUST 2009, Beijing, China, December 17-19, 2009. Revised Selected Papers*, volume 6163 of *Lecture Notes in Computer Science*, pages 156–179. Springer, 2009.

[10] Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kirli Kaynar. A logic of secure systems and its application to trusted computing. In *30th IEEE Symposium on Security and Privacy, SP 2009, 17-20 May 2009, Oakland, California, USA*, pages 221–236. IEEE Computer Society, 2009.

[11] Aurélien Francillon, Quan Nguyen, Kasper Bonne Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014.

[12] He Ge and Stephen R. Tate. A direct anonymous attestation scheme for embedded devices. In *Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Beijing, China, April 16-20, 2007, Proceedings*, volume 4450 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2007.

[13] C. Gebhardt and A. Tomlinson. Secure virtual disk images for grid computing. In *Third Asia-Pacific Trusted Infrastructure Technologies Conference*, APTC '08, pages 19–29, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.

[15] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[16] Kenneth A. Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. In *Proceedings of the 1st ACM Workshop on Scalable Trusted Computing, STC 2006, Alexandria, VA, USA, November 3, 2006*, pages 21–24. ACM, 2006.

[17] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[18] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, page 11. ACM, 2013.

[19] Intel. *Software Guard Extensions Programming Reference*, 2014. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.

[20] Jonathan Katz and Moti Yung. Scalable protocols for authenticated group key exchange. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2003.

[21] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 10:1–10:14. ACM, 2014.

[22] J. M. McCune, B. J. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, April 2008.

[23] Microsoft. *BitLocker Drive Encryption: Data Encryption Toolkit for Mobile PCs: Security Analysis*, 2007. `https://technet.microsoft.com/en-us/library/cc162804.aspx`.

[24] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 479–494. USENIX Association, 2013.

[25] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 238–252. IEEE Computer Society, 2013.

[26] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 38–54. IEEE Computer Society, 2015.

[27] Sean W. Smith. Outbound authentication for programmable secure coprocessors. *Int. J. Inf. Sec.*, 3(1):28–41, 2004.

[28] Ben Smyth, Mark Ryan, and Liqun Chen. Direct anonymous attestation (DAA): ensuring privacy with corrupt administrators. In *Security and Privacy in Ad-hoc and Sensor Networks, 4th European Workshop, ESAS 2007, Cambridge, UK, July 2-3, 2007, Proceedings*, volume 4572 of *Lecture Notes in Computer Science*, pages 218–231. Springer, 2007.

A Proof of Theorem 1

The proof is a sequence of three games presented in Figure 14 and Figure 15. The first game is simply the AC 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

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

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

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

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G_1}^{\mathsf{AC},\mathcal{A}}$ as follows: the operation of machine $\mathcal{M}_R$ 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

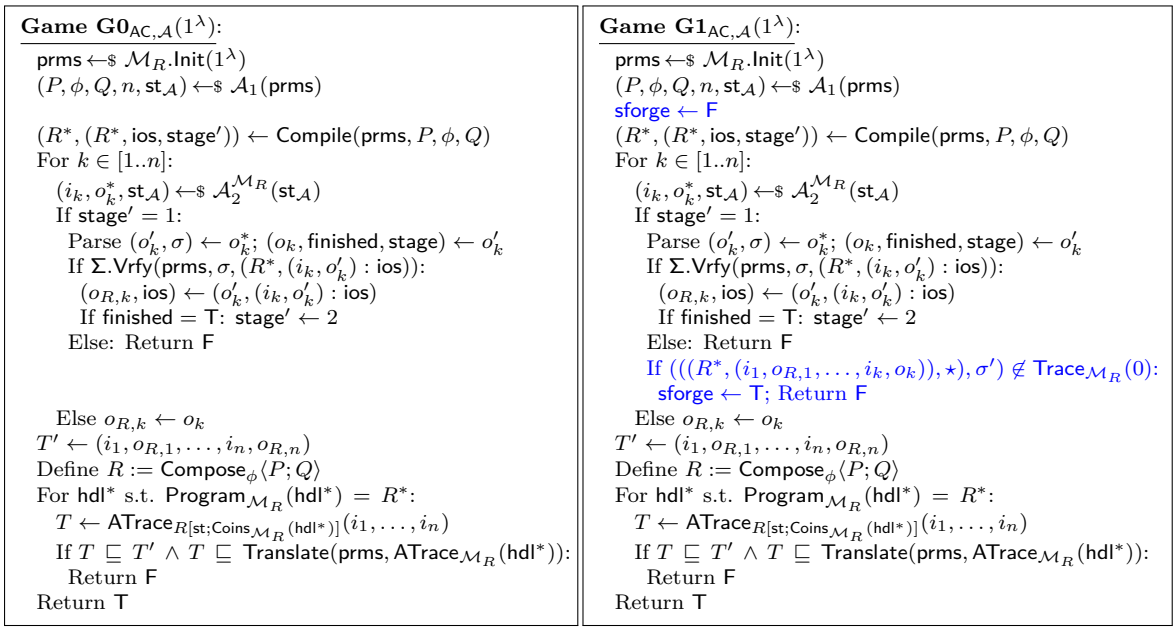

Figure 14: First game hop for the proof of security of our AC protocol.

expected to compute a signature on message m, algorithm $\mathcal{B}$ calls its own oracle on (R^*, m) to obtain σ.

When sforge is set, according to the rules of game $\mathsf{G}_1{}^{\mathsf{AC},\mathcal{A}}$, algorithm $\mathcal{B}$ outputs message (R^*, 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 $(R^*, (i_1, o_1', \ldots, i_k, o_k'))$ 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 $(R^*, (i_1, o_1', \ldots, i_k, o_k'))$ in its trace. Since we know that this is not the case, (R^*, 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 $\mathsf{G}_2{}^{\mathsf{AC},\mathcal{A}}(1^\lambda)$, 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

$$\Pr[\,\mathsf{G}_1{}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G}_2{}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{mforge}\,].$$

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 $\mathsf{G}_2{}^{\mathsf{AC},\mathcal{A}}$ as follows: the operation of machine $\mathcal{M}_R$ 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

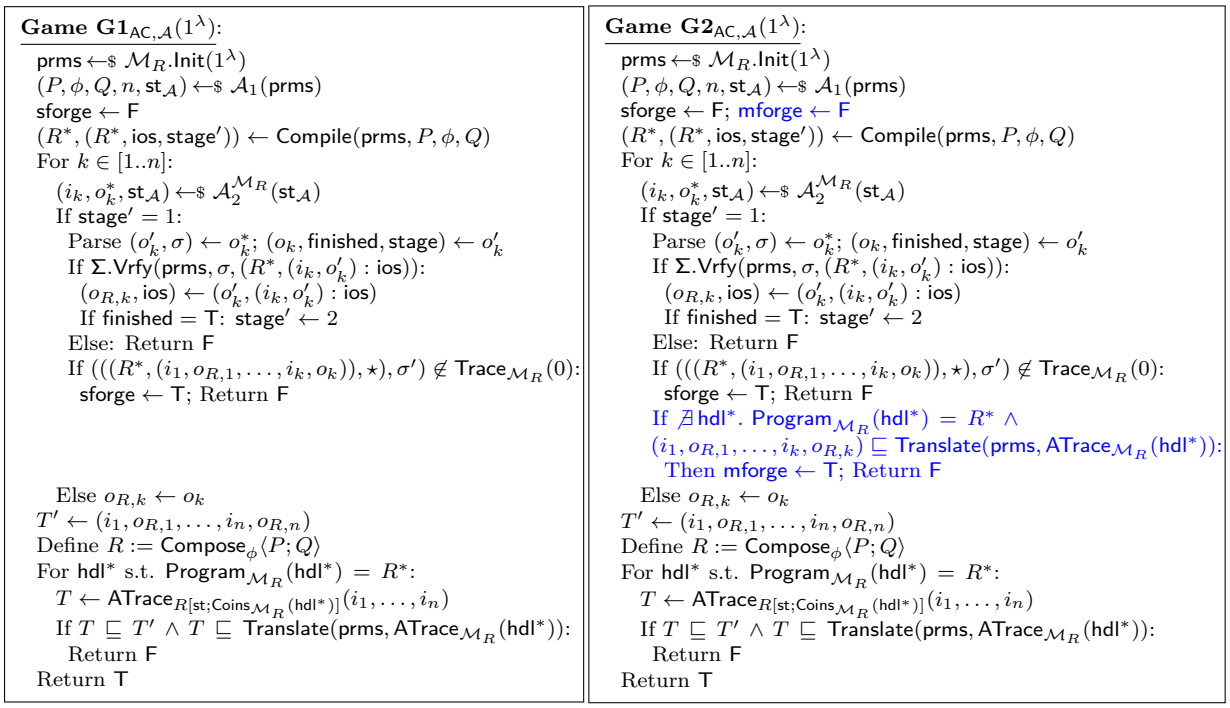

Figure 15: Second game hop for the proof of security of our AC protocol.

remote machine requests a MAC on message m from the security module, algorithm $\mathcal{C}$ calls its own oracle on (R^*, m) to obtain t.

Let $T \leftarrow (i_1, o_{R,1}, \ldots, i_k, o_{R,k})$. When mforge is set according to the rules of game $\mathsf{G}_2{}^{\mathsf{AC},\mathcal{A}}$, algorithm $\mathcal{C}$ retrieves the trace of the process with handle 0 running S^*, locates the input/output pair $(((R^*, T), \mathsf{t}), \sigma')$ and outputs message (R^*, T) and candidate tag t. To see this is a valid forgery, first observe that, having failed the sforge check, we know that $(((R^*, T), \mathsf{t}), \sigma')$ is in the trace of the process with handle 0, so by its construction we also know that the corresponding input $((R^*, T), \mathsf{t})$ must contain a valid tag. It suffices to establish that message (R^*, T) could not have been queried from the Auth oracle. Suppose that the first part of the mforge check failed, i.e., that $\nexists\, \mathsf{hdl}^*.\ \mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}^*) = R^*$. 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 R^* must have been caused by the execution of an instance of R^*. Now suppose some instances of R^* were indeed running in the remote machine, but that none of them displayed the property $(i_1, o_{R,1}, \ldots, i_k, o_{R,k}) \sqsubseteq \mathsf{Translate}(\mathsf{prms}, \mathsf{ATrace}_{\mathcal{M}_R}(\mathsf{hdl}^*))$. Then, by the construction of R^*, we can also exclude that (R^*, 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 $\mathsf{G}_2{}^{\mathsf{AC},\mathcal{A}}$. To see this, observe that when the game reaches the final check, we have the guarantee that

$$\exists\, \mathsf{hdl}^*.\ \mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}^*) = R^* \wedge$$
$$(i_1, o_{R,1}, i_2, o_{R,2}, \ldots, i_k, o_{R,k}) \sqsubseteq \mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)))$$

By the construction of R^*, it immediately follows that the final check in the game will always cause the adversary to lose:

- T is fixed by the input sequence, the value of the randomness tape and the semantics of R, which determines the sequence of outputs $(o_{1,L}, \ldots, o_{n,L})$.
- The above existential guarantee for hdl^* implies that an instance of R^* in the remote machine received the same initial k input sequence as that fixed by T.
- Since we have
$$T \leftarrow \mathsf{ATrace}_{R[\mathsf{st};\mathsf{Coins}_{\mathcal{M}_R}(\mathsf{hdl}^*)]}(i_1, \ldots, i_n)$$
we also know that the randomness tape used to produce values for T is identical to the one used in hdl^*.
- One can therefore inductively deduce, by the semantics of R^*, that the same process has produced an initial sequence of k outputs that (modulo the action of $\mathsf{Translate}$) is identical to that included in T.
- Subsequent inputs after $\mathsf{stage}' \leftarrow 2$ may produce $n - k$ additional non-attested outputs that are appended to T', and that differ from T. However, the above observation implies that $T \sqsubseteq T'$, and our security claim follows.

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 6. This is easy to ascertain given the simulator behaviour described in Figure 16: $\mathcal{S}_1$ 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 $R[\mathsf{st}](i)$ and returns an output in accordance to the behaviour of $\mathcal{M}_R$, 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. □

B Proof for Theorem 2

Our proof will follow the intuition of Katz and Yung for Theorem 1 in [20]. We will start by bounding the probabilities for the occurrence of bad events ($\mathsf{G0}^{\mathsf{AttKE},\mathcal{A}}$ to $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$), and then argue that the behaviour of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ towards an adversary is the same as the one of $\mathsf{G4}^{\Pi,\mathcal{A}}$ using a passive adversary for the original protocol Π. At this final stage we show that, given correctness and security guarantees of Π, we have a correct and secure AttKE protocol. The proof consists in a sequence of five games presented in Figures 17 to 21.

The first game is simply the AttKE security game in Figure 8 instantiated with our construction in Figure 9.

In the second game $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$, the adversary loses whenever a repeat event occurs. Intuitively, this event corresponds to the adversary generating two sessions with the same key pair, and hence constitutes a forgery with respect to Σ. Given that the two games are identical until this event occurs, we have that

$$\Pr[\,\mathsf{G0}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{repeat}\,].$$

Let q be the maximum number of calls to $\mathsf{NewLocal}$ allowed. We upper bound the distance between these two games, by constructing an adversary $\mathcal{B}$ against the existential unforgeability of signature scheme Σ such that

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

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

$\underline{\mathcal{S}_2(\mathsf{hdl}, P, \phi, Q, R, i, o^*, \mathsf{st}_\mathcal{S})}$:
Parse $(\mathsf{key}, \mathsf{sk}, \mathsf{Traces}) \leftarrow \mathsf{st}_\mathcal{S}$
If $\nexists\,(\mathsf{ios}, \mathsf{stage}) \in \mathsf{Traces}[\mathsf{hdl}]$: $\mathsf{ios} \leftarrow [\,]$; $\mathsf{stage} \leftarrow 1$
$\mathsf{ios} \leftarrow (i, o^*) : \mathsf{ios}$
If $\mathsf{stage} = 1$: $\mathsf{m} \leftarrow (o^*, \Pi.\mathsf{Mac}(\mathsf{key}, R, \mathsf{ios}))$
Else $\mathsf{m} \leftarrow o^*$
Parse $(o, \mathsf{finished}, \mathsf{stage}) \leftarrow o^*$
If $\mathsf{finished} = \mathsf{T}$: $\mathsf{stage} \leftarrow 2$
$\mathsf{Traces}[\mathsf{hdl}] \leftarrow (\mathsf{ios}, \mathsf{stage})$; $\mathsf{st}_\mathcal{S} \leftarrow (\mathsf{key}, \mathsf{sk}, \mathsf{Traces})$
Return $(\mathsf{m}, \mathsf{st}_\mathcal{S})$

- $\mathcal{S}_3$ standardly computes the next output given input i, program R and state st. The result is afterwards treated similar to $\mathcal{S}_2$.

$\underline{\mathcal{S}_3(\mathsf{hdl}, R, i, \mathsf{st}, \mathsf{st}_\mathcal{S})}$:
Parse $(\mathsf{key}, \mathsf{sk}, \mathsf{Traces}) \leftarrow \mathsf{st}_\mathcal{S}$
If $\mathsf{hdl} = 0$:
  Parse $(i, \mathsf{t}) \leftarrow i^*$
  If $\Pi.\mathsf{Ver}(\mathsf{key}, \mathsf{t}, i)$: Return $\Sigma.\mathsf{Sign}(\mathsf{sk}, i)$
  Else Return $\perp$
If $\nexists\,(\mathsf{ios}, \mathsf{stage}) \in \mathsf{Traces}[\mathsf{hdl}]$: $\mathsf{ios} \leftarrow [\,]$; $\mathsf{stage} \leftarrow 1$
$o^* \leftarrow\$\ R[\mathsf{st}](i)$
If $\mathsf{stage} = 1$: $\mathsf{m} \leftarrow (o^*, \Pi.\mathsf{Mac}(\mathsf{key}, R, \mathsf{st.ios}))$
Else $\mathsf{m} \leftarrow o^*$
Parse $(o, \mathsf{finished}, \mathsf{stage}) \leftarrow o^*$
If $\mathsf{finished} = \mathsf{T}$: $\mathsf{stage} \leftarrow 2$
$\mathsf{Traces}[\mathsf{hdl}] \leftarrow (\mathsf{ios}, \mathsf{stage})$; $\mathsf{st}_\mathcal{S} \leftarrow (\mathsf{key}, \mathsf{sk}, \mathsf{Traces})$
Return $(\mathsf{m}, \mathsf{st}_\mathcal{S})$

Figure 16: Description of simulator $\mathcal{S}$

$$\Pr[\,\mathsf{repeat}\,] \leq \frac{\mathsf{Adv}^{\mathsf{UF}}_{\Sigma,\mathcal{B}}(\lambda) * \mathsf{q}}{2}.$$

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$ as follows: at the beginning of the game, $\mathcal{B}$ has to try and guess which session will have a duplicate key. As such, it samples uniformly from $[1..\mathsf{q}]$ a session s and replaces the public key generated by $\Sigma.\mathsf{Gen}$ in $\mathsf{NewLocal}$ for instance $i = s$ with the public key $\mathsf{pk}_{\mathsf{UF}}$ provided by $\mathsf{UF}^{\Sigma,\mathcal{B}}$. Every time instance s has to produce a signature in $\mathsf{SendLocal}(\mathsf{m}, s)$, instead of $\Sigma.\mathsf{Sign}(\mathsf{sk}, t)$, $\mathcal{B}$ calls Oracle $\mathsf{Sign}(t)$. Additionally, $\mathcal{B}$ will store all key pairs and session ids generated by $\mathsf{NewLocal}$ s.t. $i \neq s$ in a list keys. When

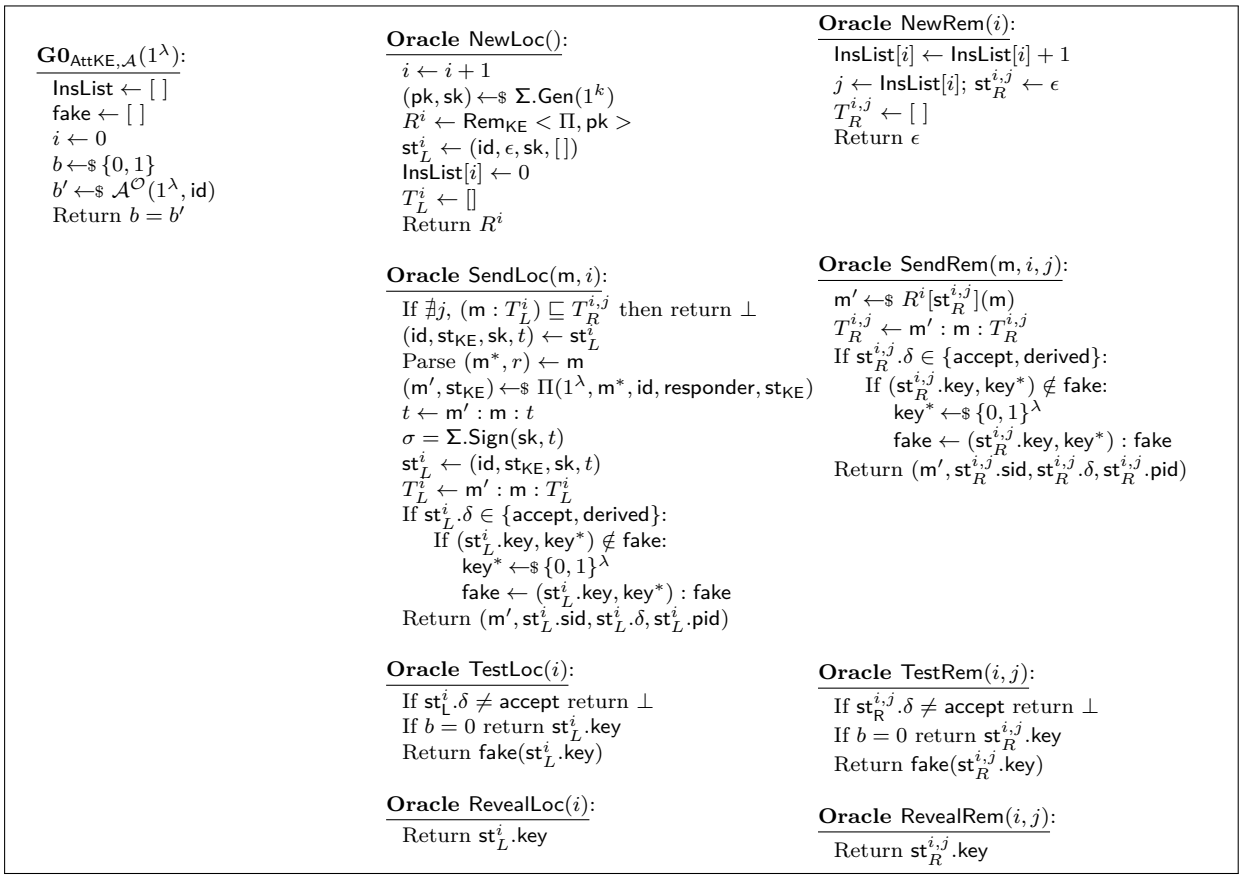

Figure 17: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

the game terminates, the adversary looks up on list keys for a key pair with public key $\mathsf{pk}_{\mathsf{UF}}$, extracts the associated secret key sk, executes $\sigma \leftarrow\$ \Sigma.\mathsf{Sign}(\mathsf{sk}, \mathsf{m}')$ for any m' not yet queried to Oracle Sign and presents (m', σ) as a challenge for $\mathsf{UF}^{\Sigma,\mathcal{B}}$. It remains to show that, when repeat is set, $\mathcal{B}$ wins $\mathsf{UF}^{\Sigma,\mathcal{B}}$ with probability $2/\mathsf{q}$

When the game ends and $\mathsf{repeat} = \mathsf{T}$, we have at least two duplicate R. This implies that $\exists i, j$ s.t. $(\mathsf{pk}_i, \mathsf{sk}_i) = (\mathsf{pk}_j, \mathsf{sk}_j), i \neq j$. If it is the case that $i = s \vee j = s$, $\mathcal{B}$ has either sk_i or sk_j in keys, and can use that to generate the signature that wins the $\mathsf{UF}^{\Sigma,\mathcal{B}}$ game. Since $i, j \in [1..\mathsf{q}]$ and s is sampled uniformly from $[1..\mathsf{q}]$, we have that this happens with probability $2/\mathsf{q}$, and we therefore conclude that $\mathcal{B}$ outputs a valid forgery with the same probability whenever repeat occurs.

In game $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$, the adversary loses whenever a rnonce event occurs. Intuitively, this event corresponds to the adversary generating two duplicate nonces. Given that the two games are identical until this event occurs, we have that

$$\Pr[\,\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{rnonce}\,].$$

Let q_R be the number of calls to $\mathsf{NewRemote}$ allowed to the adversary. We upper bound the distance between these two games such that

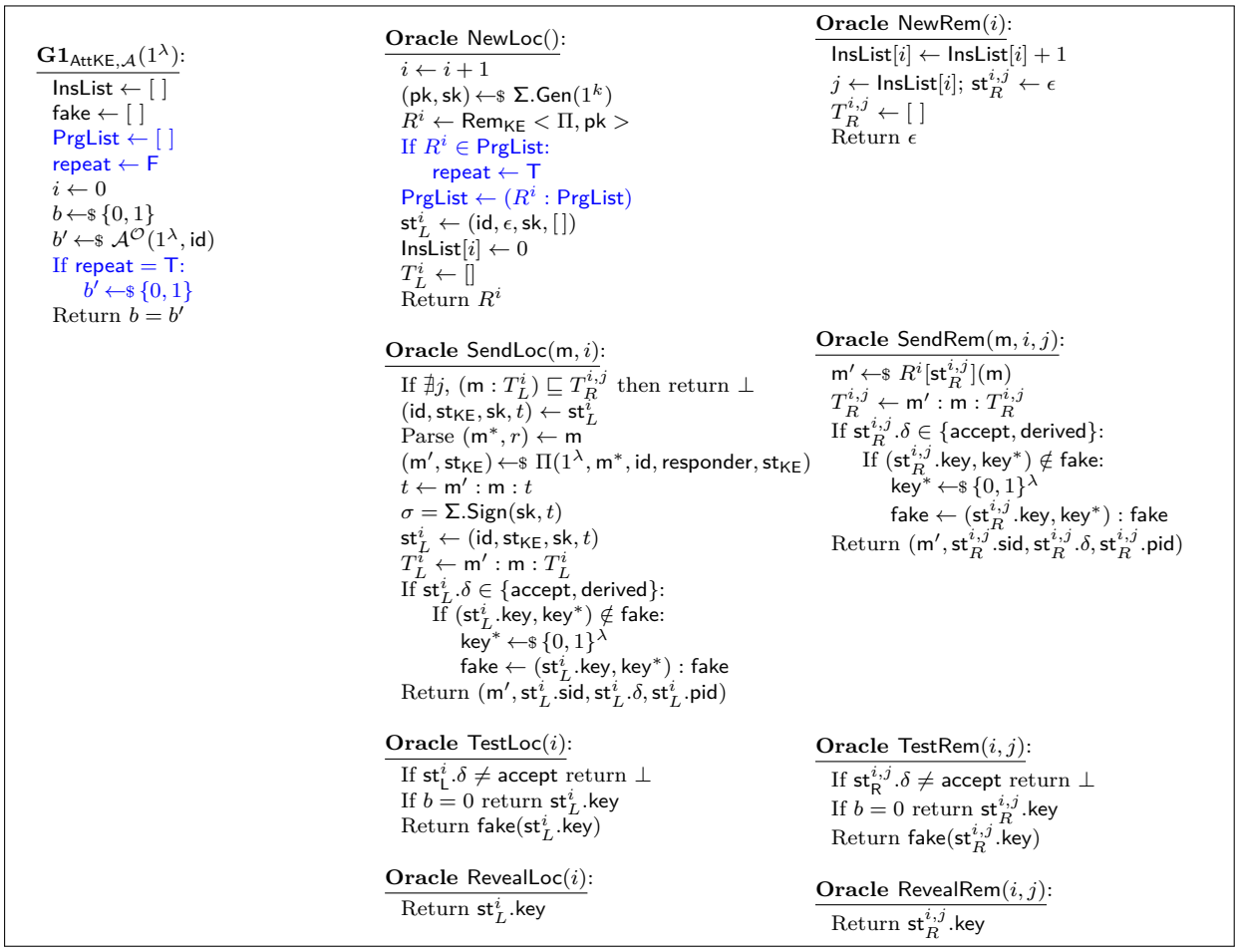

Figure 18: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

$$\Pr[\,\mathsf{rnonce}\,] \leq \frac{\mathsf{q}_R^2}{2^k}\,.$$

From the rules of $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$ and the construction of R, we know that a new r is generated at every query of SendRemote such that $T^{i,j}_R = [\,]$. This only happens at most once for every new $T^{i,j}_R$, i.e., at most once for every call of NewRemote. Since r is sampled uniformly from a subset of $\{0,1\}^k$, we conclude that the probability of rnonce is $\mathsf{q}_R^2/2^k$.

In game $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$, the adversary loses whenever a forge event occurs. Intuitively, this event corresponds to the adversary producing a signature that was not computed by SendLocal, and hence constitutes a forgery with respect to Σ. Given that the two games are identical until this event occurs, we have that

$$\Pr[\,\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{forge}\,]\,.$$

Let q be the maximum number of calls to NewLocal allowed. We upper bound the distance between these two games, by constructing an adversary $\mathcal{C}$ against the existential unforgeability of signature scheme Σ such that

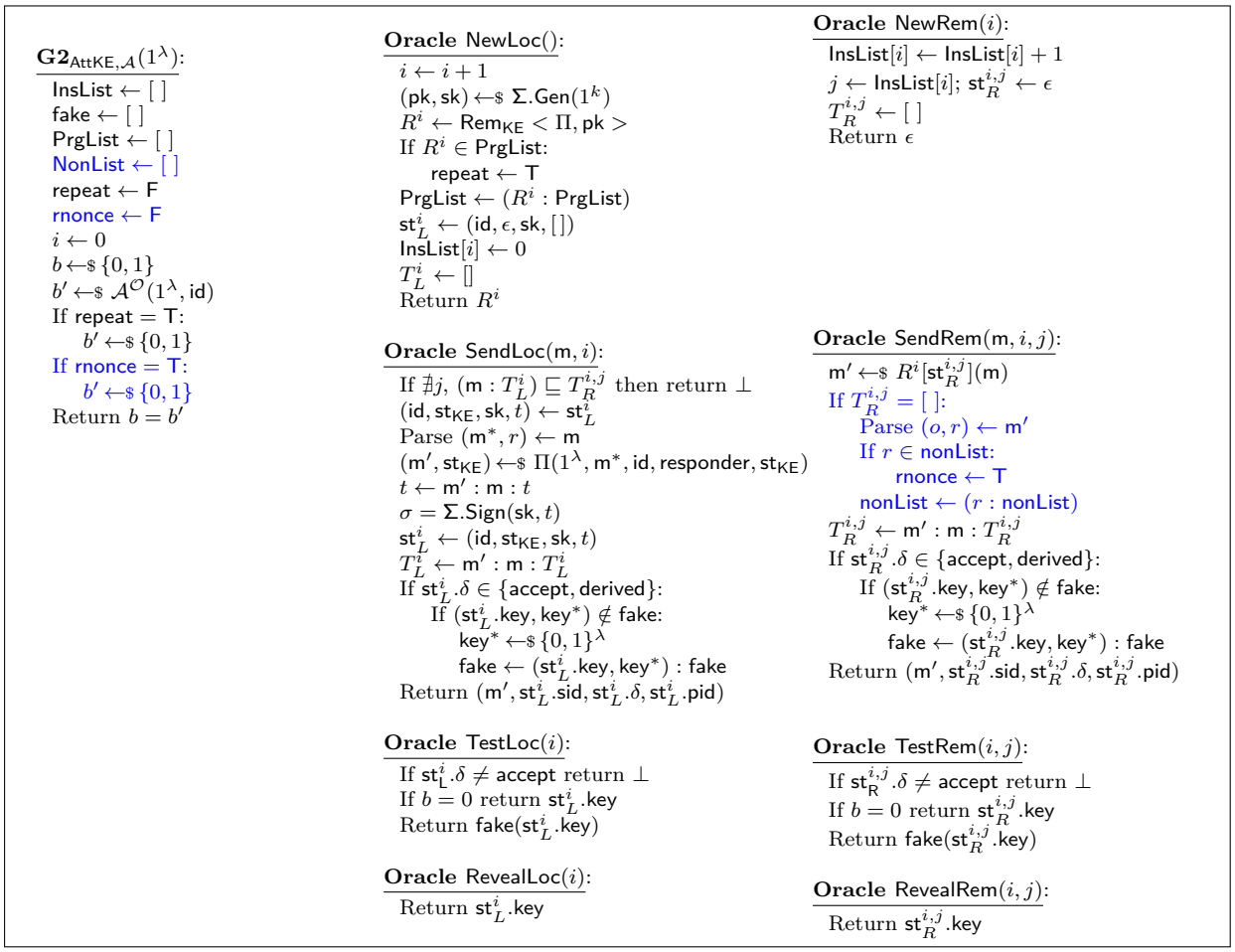

Figure 19: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

$$\Pr[\,\mathsf{forge}\,] \;\leq\; \mathsf{Adv}^{\mathsf{UF}}_{\Sigma,\mathcal{C}}(\lambda) * \mathsf{q}\,.$$

Adversary $\mathcal{C}$ simulates the environment of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ as follows: at the beginning of the game, $\mathcal{C}$ has to try and guess which session will have a duplicate key. As such, it samples uniformly from [1..q] a session s and replaces the public key generated by $\Sigma.\mathsf{Gen}$ in NewLocal for instance $i = s$ with the public key $\mathsf{pk}_{\mathsf{UF}}$ provided by $\mathsf{UF}^{\Sigma,\mathcal{C}}$. Every time instance s has to produce a signature in $\mathsf{SendLocal}(\mathsf{m}, s)$, instead of $\Sigma.\mathsf{Sign}(\mathsf{sk}, t)$, $\mathcal{C}$ calls Oracle $\mathsf{Sign}(t)$. During the execution of $\mathsf{SendRemote}(\mathsf{m}, i, j)$ that sets $\mathsf{forge} = \mathsf{T}$, $\mathcal{C}$ presents $((o : \mathsf{st}^{i,j}_R.t), \sigma)$ as a challenge for $\mathsf{UF}^{\Sigma,\mathcal{C}}$. It remains to show that, when forge is set, $\mathcal{C}$ wins $\mathsf{UF}^{\Sigma,\mathcal{C}}$ with probability $1/\mathsf{q}$.

When $\mathsf{forge} = \mathsf{T}$ we have that, for some execution of $\mathsf{SendRemote}(\mathsf{m}, i, j)$, $\mathsf{st}^{i,j}_R.\delta \neq \mathsf{reject} \wedge (o : \mathsf{st}^{i,j}_R) \notin \mathsf{sigList}$. From the construction of R, we know that $\mathsf{st}^{i,j}_R.\delta \neq \mathsf{reject}$ implies that the provided $((o : \mathsf{st}^{i,j}_R.t), \sigma)$ is a valid message/signature pair for session i. If $i = s$, this is also a valid message/signature pair for $\mathsf{UF}^{\Sigma,\mathcal{C}}$. Now observe that $(o : \mathsf{st}^{s,j}_R.t) \notin \mathsf{sigList}$ assures us that $(o : \mathsf{st}^{s,j}_R.t)$ was not queried to oracle Sign: rnonce establishes that every nonce is unique, and every execution of SendLocal adds the nonce r to the signed message, so every call to oracle Sign is unique. From

$$\text{If } \nexists j,\ (\mathsf{m} : T_L^s) \sqsubseteq T_R^{s,j} \text{ then return } \perp$$

and repeat, we know that if oracle Sign was called for some $(m' : \mathsf{st}_R^{s,j}.t)$, then SendLocal of session s would be responding to the sequence of messages exchanged with the unique instance j, matching $T_R^{s,j}$ that coincides (modulo signatures) with $\mathsf{st}_R^{s,j}.t$. However, by the construction of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$, that would imply $\mathsf{sigList} \leftarrow (\mathsf{st}_R^{s,j}.t : \mathsf{sigList})$, and we know that $\mathsf{st}_R^{s,j}.t \notin \mathsf{sigList}$. We therefore conclude that $(\mathsf{st}_R^{i,j}.t, \sigma)$ is a winning output for game $\mathsf{UF}^{\Sigma,\mathcal{C}}$ if $i = s$. That probability is $1/\mathsf{q}$.

$\mathbf{G3}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda)$:
- $\mathsf{InsList} \leftarrow [\,]$
- $\mathsf{fake} \leftarrow [\,]$
- $\mathsf{PrgList} \leftarrow [\,]$
- $\mathsf{NonList} \leftarrow [\,]$
- $\mathsf{sigList} \leftarrow [\,]$
- $\mathsf{repeat} \leftarrow \mathsf{F}$
- $\mathsf{rnonce} \leftarrow \mathsf{F}$
- $\mathsf{forge} \leftarrow \mathsf{F}$
- $i \leftarrow 0$
- $b \leftarrow\$ \{0,1\}$
- $b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(1^\lambda, \mathsf{id})$
- If $\mathsf{repeat} = \mathsf{T}$:
  - $b' \leftarrow\$ \{0,1\}$
- If $\mathsf{rnonce} = \mathsf{T}$:
  - $b' \leftarrow\$ \{0,1\}$
- If $\mathsf{forge} = \mathsf{T}$:
  - $b' \leftarrow\$ \{0,1\}$
- Return $b = b'$

Oracle NewLoc():
- $i \leftarrow i + 1$
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow\$\ \Sigma.\mathsf{Gen}(1^k)$
- $R^i \leftarrow \mathsf{Rem}_{\mathsf{KE}} < \Pi, \mathsf{pk} >$
- If $R^i \in \mathsf{PrgList}$:
  - $\mathsf{repeat} \leftarrow \mathsf{T}$
- $\mathsf{PrgList} \leftarrow (R^i : \mathsf{PrgList})$
- $\mathsf{st}_L^i \leftarrow (\mathsf{id}, \epsilon, \mathsf{sk}, [\,])$
- $\mathsf{InsList}[i] \leftarrow 0$
- $T_L^i \leftarrow []$
- Return R^i

Oracle SendLoc(m, i):
- If $\nexists j,\ (\mathsf{m} : T_L^i) \sqsubseteq T_R^{i,j}$ then return $\perp$
- $(\mathsf{id}, \mathsf{st}_{\mathsf{KE}}, \mathsf{sk}, t) \leftarrow \mathsf{st}_L^i$
- Parse $(\mathsf{m}^*, r) \leftarrow \mathsf{m}$
- $(\mathsf{m}', \mathsf{st}_{\mathsf{KE}}) \leftarrow\$\ \Pi(1^\lambda, \mathsf{m}^*, \mathsf{id}, \mathsf{responder}, \mathsf{st}_{\mathsf{KE}})$
- $t \leftarrow \mathsf{m}' : \mathsf{m} : t$
- $\sigma = \Sigma.\mathsf{Sign}(\mathsf{sk}, t)$
- $\mathsf{sigList} \leftarrow t : \mathsf{sigList}$
- $\mathsf{st}_L^i \leftarrow (\mathsf{id}, \mathsf{st}_{\mathsf{KE}}, \mathsf{sk}, t)$
- $T_L^i \leftarrow \mathsf{m}' : \mathsf{m} : T_L^i$
- If $\mathsf{st}_L^i.\delta \in \{\mathsf{accept}, \mathsf{derived}\}$:
  - If $(\mathsf{st}_L^i.\mathsf{key}, \mathsf{key}^*) \notin \mathsf{fake}$:
    - $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    - $\mathsf{fake} \leftarrow (\mathsf{st}_L^i.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
- Return $(\mathsf{m}', \mathsf{st}_L^i.\mathsf{sid}, \mathsf{st}_L^i.\delta, \mathsf{st}_L^i.\mathsf{pid})$

Oracle TestLoc(i):
- If $\mathsf{st}_{\mathsf{L}}^i.\delta \neq \mathsf{accept}$ return $\perp$
- If $b = 0$ return $\mathsf{st}_L^i.\mathsf{key}$
- Return $\mathsf{fake}(\mathsf{st}_L^i.\mathsf{key})$

Oracle RevealLoc(i):
- Return $\mathsf{st}_L^i.\mathsf{key}$

Oracle NewRem(i):
- $\mathsf{InsList}[i] \leftarrow \mathsf{InsList}[i] + 1$
- $j \leftarrow \mathsf{InsList}[i]$; $\mathsf{st}_R^{i,j} \leftarrow \epsilon$
- $T_R^{i,j} \leftarrow [\,]$
- Return ϵ

Oracle SendRem(m, i, j):
- $\mathsf{m}' \leftarrow\$\ R^i[\mathsf{st}_R^{i,j}](\mathsf{m})$
- If $T_R^{i,j} = [\,]$:
  - Parse $(o, r) \leftarrow \mathsf{m}'$
  - If $r \in \mathsf{nonList}$:
    - $\mathsf{rnonce} \leftarrow \mathsf{T}$
  - $\mathsf{nonList} \leftarrow (r : \mathsf{nonList})$
- Else:
  - Parse $(o, \sigma) \leftarrow \mathsf{m}$
  - If $\mathsf{st}_R^{i,j}.\delta \neq \mathsf{reject} \ \wedge (o : \mathsf{st}_R^{i,j}.t) \notin \mathsf{sigList}$
    - $\mathsf{forge} \leftarrow \mathsf{T}$
- $T_R^{i,j} \leftarrow \mathsf{m}' : \mathsf{m} : T_R^{i,j}$
- If $\mathsf{st}_R^{i,j}.\delta \in \{\mathsf{accept}, \mathsf{derived}\}$:
  - If $(\mathsf{st}_R^{i,j}.\mathsf{key}, \mathsf{key}^*) \notin \mathsf{fake}$:
    - $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    - $\mathsf{fake} \leftarrow (\mathsf{st}_R^{i,j}.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
- Return $(\mathsf{m}', \mathsf{st}_R^{i,j}.\mathsf{sid}, \mathsf{st}_R^{i,j}.\delta, \mathsf{st}_R^{i,j}.\mathsf{pid})$

Oracle TestRem(i, j):
- If $\mathsf{st}_{\mathsf{R}}^{i,j}.\delta \neq \mathsf{accept}$ return $\perp$
- If $b = 0$ return $\mathsf{st}_R^{i,j}.\mathsf{key}$
- Return $\mathsf{fake}(\mathsf{st}_R^{i,j}.\mathsf{key})$

Oracle RevealRem(i, j):
- Return $\mathsf{st}_R^{i,j}.\mathsf{key}$

Figure 20: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

Finally, in game $\mathsf{G4}^{\Pi,\mathcal{A}}$, we no longer require AttKE, but instead make use of a passive adversary and its corresponding oracles $\{\mathsf{Execute}(i,j), \mathsf{Reveal}(i,s), \mathsf{Test}(i,s)\}$. The intuition is that, at this fifth game, all calls to the compiled AttKE protocol can be perfectly simulated using the original protocol Π. More formally, we want to show that

$$\Pr[\,\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] = \Pr[\,\mathsf{G4}^{\Pi,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,].$$

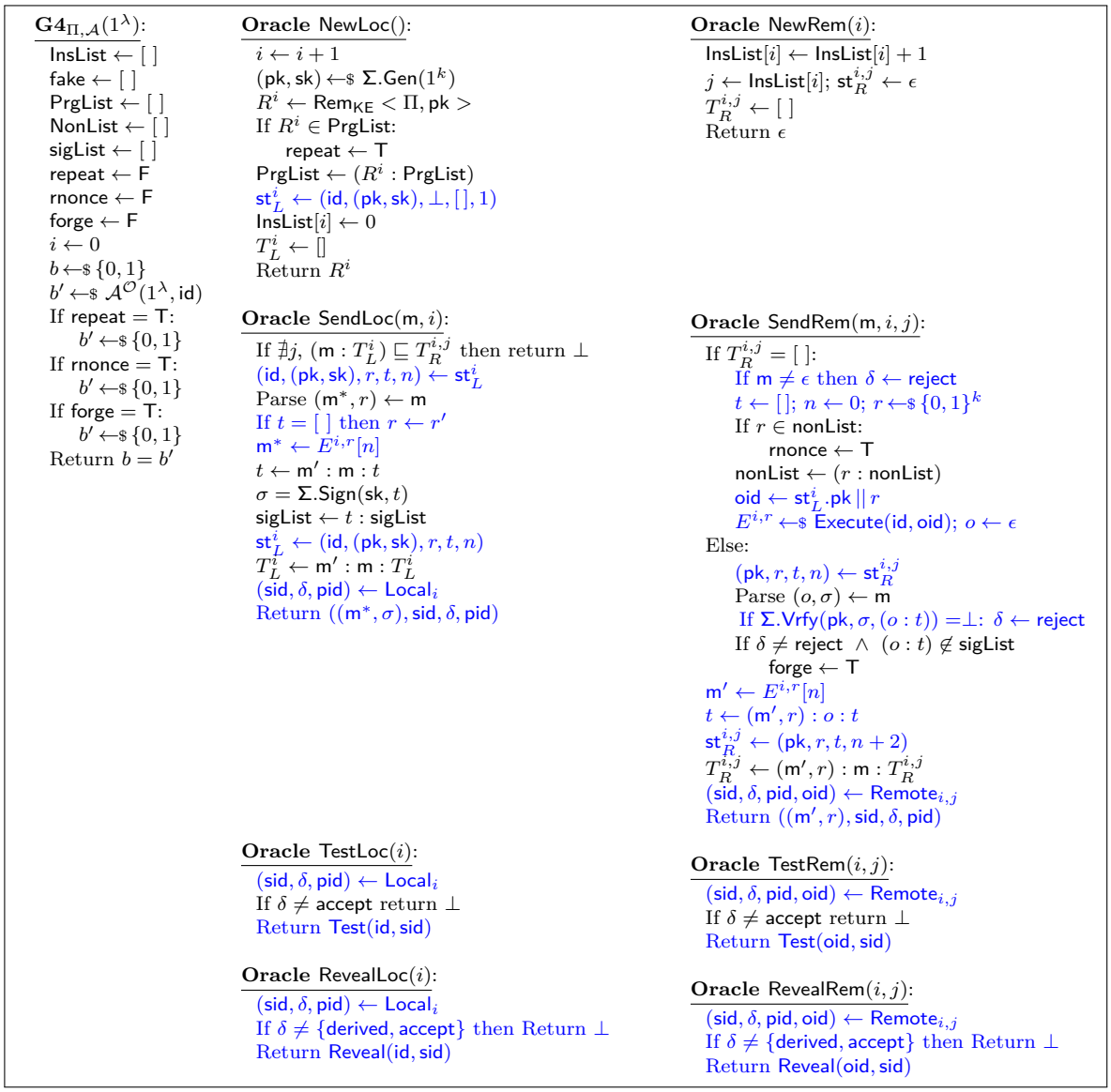

Figure 21: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

Whenever $\mathsf{G4}^{\Pi,\mathcal{A}}$ is required to respond to SendLocal or SendRemote, instead of executing Π and R (respectively), it will follow a unique Execute(id, oid) that is associated with sessions i and nonce r. Additionally, $\mathsf{G4}^{\Pi,\mathcal{A}}$ tracks when any execution should be rejected, and responds to Reveal/Test with either $\perp$ or with the output of the same oracles from the passive adversary. To help with this, $\mathsf{G4}^{\Pi,\mathcal{A}}$ stores a list of Execute transcripts $E^{i,r}$. Messages exchanged from such transcripts will be tracked locally and remotely with a counter n. From the information in $(\mathsf{st}^i_L, \mathsf{st}^{i,j}_R)$ and transcripts $E^{i,r}$ we assume that it is possible to infer $(\mathsf{sid}, \delta, \mathsf{pid})$ and $(\mathsf{sid}, \delta, \mathsf{pid}, \mathsf{oid})$ via Local_i and $\mathsf{Remote}_{i,j}$, respectively.

The changes in NewLocal merely reflect additional information stored in existing structures. In SendLocal, $E^{i,r}[n]$ is used instead of Π to provide the next message. In SendRemote, instead of running R, the specification in Figure 9 is followed on the oracle itself, with the

exception of calling $E^{i,j}[n]$ instead of using Π to provide the next message. In $\mathsf{Reveal}/\mathsf{Test}$, either $\perp$ is returned or the result of another oracle $\mathsf{Reveal}/\mathsf{Test}$ is given, instead of directly providing a key. In these scenarios, handling of fake keys is delegated to Test queries of the passive adversary. As such, to validate these games as equivalent, we must show that all calls to $E^{i,r}$ correspond to the Π replaced, and that $\mathsf{Reveal}/\mathsf{Test}$ are responding similarly to $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$.

Observe that, by the construction of $E^{i,r} \leftarrow\!\!\$\ \mathsf{Execute}(\mathsf{id},\mathsf{oid})$,

$$E^{i,r} = [o_1 \leftarrow\!\!\$\ \Pi(1^\lambda, \epsilon, \mathsf{oid}, \mathsf{initiator}, st_r), o_2 \leftarrow\!\!\$\ \Pi(1^\lambda, o_1, \mathsf{id}, \mathsf{responder}, st_i), \ldots]$$

the transcript $E^{i,r}$ contains a list of specific executions of $\Pi(1^\lambda, \mathsf{m}, i, \rho, st)$. We must show that every $E^{i,j}[n]$ matches to the output of $\Pi(1^\lambda, \mathsf{m}, i, \rho, st)$ executed in $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$. First note that, from repeat, we know that every i is associated with a different R. This means that every $\mathsf{SendRemote}(\mathsf{m}, i, j)$ will be associated with a unique st_L^i. Many executions of the same session i may occur, so the nonce associated cannot be determined at $\mathsf{NewLocal}$ (hence $\mathsf{st}_L^i \leftarrow (\mathsf{id}, (\mathsf{pk}, \mathsf{sk}), \perp, [\,], 1)$), but we know that every first message of AttKE will fix a unique r, so $E^{i,r}$ can be established in the first non-reject call of $\mathsf{SendLocal}(\mathsf{m}, i)$, and a 1-to-1 relation with instance j is given by

$$\text{If } \nexists j,\ (m : T_L^i) \sqsubseteq T_R^{i,j} \text{ then return } \perp \tag{1}$$

and forge. Let $\mathsf{SendLocal}_k$ and $\mathsf{SendRemote}_k$ be the k-th execution of these Oracles.

- $\mathsf{SendRemote}_1(\mathsf{m}, i, j)$: If $\delta \neq \mathsf{reject}$, $r \leftarrow\!\!\$\ \{0,1\}^k$, $[\Pi(1^\lambda, \epsilon, \mathsf{pk}||r, \mathsf{initiator}, \mathsf{st}), \ldots] \leftarrow\!\!\$\ \mathsf{Execute}(\mathsf{id}, \mathsf{pk}||r)$ and we have that $\Pi(1^\lambda, \epsilon, \mathsf{pk}||r, \mathsf{initiator}, \mathsf{st}) = E^{i,r}[0] = \mathsf{m}_1$; $\mathsf{st}_R^{i,j}.t = [(\mathsf{m}_1, r), \epsilon]$; $\mathsf{st}_R^{i,j}.n = 2$.

- $\mathsf{SendLocal}_2(\mathsf{m}, i)$: If $\mathsf{st}_L^i.\delta \neq \mathsf{reject}$, we know that $((o, r) : \mathsf{st}_L^i.t)$ is in the prefix trace of a unique instance $\mathsf{SendRemote}_1$ (from (1)). Since $\mathsf{st}_L^i.t = [\,]$, m was the first message produced by $\mathsf{SendRemote}_1$, so we can set r as the nonce of the execution, retrieving the second message from $\mathsf{Execute}(\mathsf{id}, \mathsf{pk}||r)$ in $E^{i,r}$, the unique transcript between session i and the remote j that produced r. As such, we have that $\Pi(1^\lambda, \mathsf{m}', \mathsf{id}, \mathsf{responder}, \mathsf{st}) = E^{i,r}[1] = \mathsf{m}_2$. $\mathsf{st}_L^i.t = [\mathsf{m}_2, (\mathsf{m}_1, r'), \epsilon]$; $\mathsf{st}_L^i.n \leftarrow 3$.

- $\mathsf{SendRemote}_n(\mathsf{m}, i, j)$: If $\delta \neq \mathsf{reject}$, Parse $(o, \sigma) \leftarrow \mathsf{m}$; $\Sigma.\mathsf{Vrfy}(\mathsf{pk}, \sigma, (o : \mathsf{st}_R^{i,j}.t)) \neq \perp$ and $(o : \mathsf{st}_R^{i,j}.t) \in \mathsf{sigList}$ means that m was the $(n-1)$-th message produced by instance $\mathsf{SendLocal}_{n-1}(\mathsf{m}_{n-2}, i)$ with $\mathsf{st}_L^i.t = [o, \ldots, (\mathsf{m}_1, r), \epsilon]$, following $(o : \mathsf{st}_R^{i,j}.t)$ (from forge). $(o : \mathsf{st}_R^{i,j}.t)$ was constructed with the first $n-1$ messages in the transcript of $\mathsf{Execute}(\mathsf{id}, \mathsf{pk}||r)$: $E^{i,r}$, so given that $E^{i,r}[n]$ provides the n-th message of $E^{i,r}$, we have that $\Pi(1^\lambda, o, \mathsf{pk}||r, \mathsf{initiator}, \mathsf{st}) = E^{i,r}[n]$. $\mathsf{st}_R^{i,j}.n = n + 2$.

- $\mathsf{SendLocal}_n(\mathsf{m}, i)$: If $\mathsf{st}_L^i.\delta \neq \mathsf{reject}$, we know that m was the $(n-1)$-th message produced by some instance $\mathsf{SendRemote}_{n-1}$ with $([\mathsf{m}, \ldots, \mathsf{m}_2, (\mathsf{m}_1, r), \epsilon])$ in its trace $\mathsf{st}_R^{i,j}.t$ (from (1)). $\mathsf{st}_R^{i,j}.t$ was also constructed with the first $n-1$ messages in the transcript of $\mathsf{Execute}(\mathsf{id}, \mathsf{pk}||r)$: $E^{i,r}$, so given that $E^{i,r}[n]$ provides the n-th message of $E^{i,r}$, we have that $\Pi(1^\lambda, m', \mathsf{id}, \mathsf{responder}, \mathsf{st}) = E^{i,r}[n]$. $\mathsf{st}_L^i.n = n + 2$.

Regarding Reveal and Test queries, observe that either the instances of i or j have $\delta \neq \{\mathsf{derived}, \mathsf{accept}\}$ or $\delta \neq \mathsf{accept}$, respectively, which given the previous is the same in both

scenarios, or $\mathcal{A}$ gets the response from the Oracle of the passive adversary regarding the unique $\mathsf{Execute}(\mathsf{id}, \mathsf{oid})$ associated with sessions i and j, and the output is also the same in both scenarios.

As such, we have that the behaviour of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ towards an adversary $\mathcal{A}$ is indistinguishable to the one provided in $\mathsf{G4}^{\Pi,\mathcal{A}}$ interacting with a passive adversary for the original protocol. This provides us with the correctness and security guarantees of the passive protocol Π modulo any attacks that would also be possible in the passive scenario. Now observe that, given that rnonce assures unique values for oid, and given that $\mathsf{Local}_{\mathsf{KE}}$ and $\mathsf{Remote}_{\mathsf{KE}}$ have $\rho = \mathsf{responder}$ and $\rho = \mathsf{initiator}$, respectively, it is trivial to infer that this implies the correctness and security of AttKE.

To conclude, we have that

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AttKE},\mathcal{A}} &= \Pr[\,\mathsf{G}_0^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)\,] - \Pr[\,\mathsf{G}_4^{\Pi,\mathcal{A}}(1^\lambda)\,] \\
&= (\sum_{i=0}^{3} \Pr[\,\mathsf{G}_i^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)\,] - \Pr[\,\mathsf{G}_{i+1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)]) + (\Pr[\,\mathsf{G}_3^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)] - \Pr[\,\mathsf{G}_4^{\Pi,\mathcal{A}}(1^\lambda)]) \\
&\leq \Pr[\mathsf{repeat}] + \Pr[\mathsf{rnonce}] + \Pr[\mathsf{forge}] + \mathsf{Adv}^{\mathsf{Att}}_{\Pi,\mathcal{A}}(\lambda) \\
&\leq \frac{3 * \mathsf{Adv}^{\mathsf{UF}}_{\Sigma,\mathcal{D}}(\lambda) * \mathsf{q}}{2} + \frac{\mathsf{q}_R^2}{2^k} + \mathsf{Adv}^{\mathsf{Att}}_{\Pi,\mathcal{A}}(\lambda)
\end{aligned}$$

and Theorem 2 follows.

□

C Proof for Theorem 3

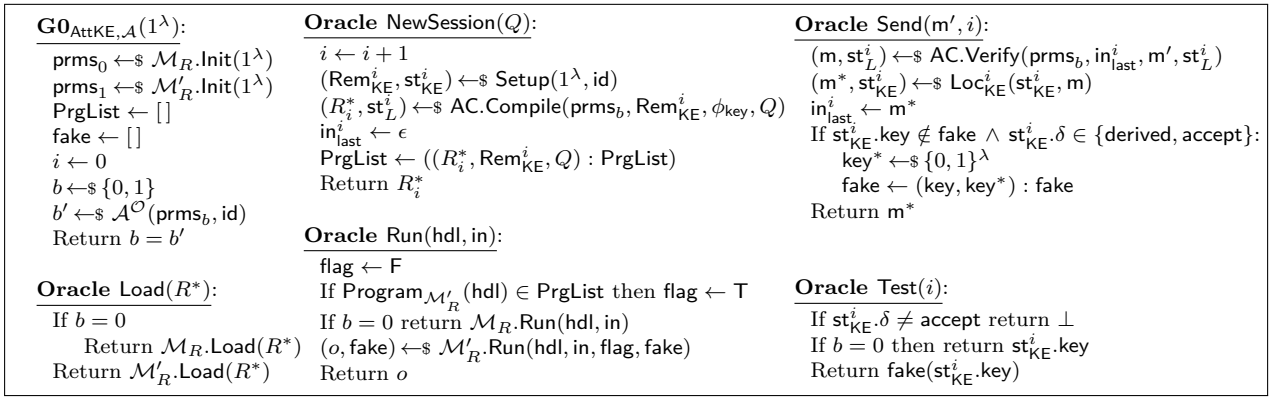

Figure 22: Game defining the utility of an AttKE scheme when used in the context of attested computation.

In this proof, we will start by bounding the possibility for the occurrence of a bad event ($\mathsf{G0}^{\mathsf{AC},\mathcal{A}}$ to $\mathsf{G1}^{\mathsf{AC},\mathcal{A}}$). We will then replace the machine execution with the indistinguishable behavior of the simulator of minimum leakage game in Figure 6 for $\mathsf{G2}^{\mathsf{AC},\mathcal{A}}$. Finally, we argue that, given these circumstances, this scenario is the same as one of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ using the oracles of key exchange for attested computation in Figure 8 modulo any advantage the adversary may gain from the AttKE scheme. The proof consists in a sequence of four games presented in figures 22 to 25. The first game is simply the utility game in Figure 10.

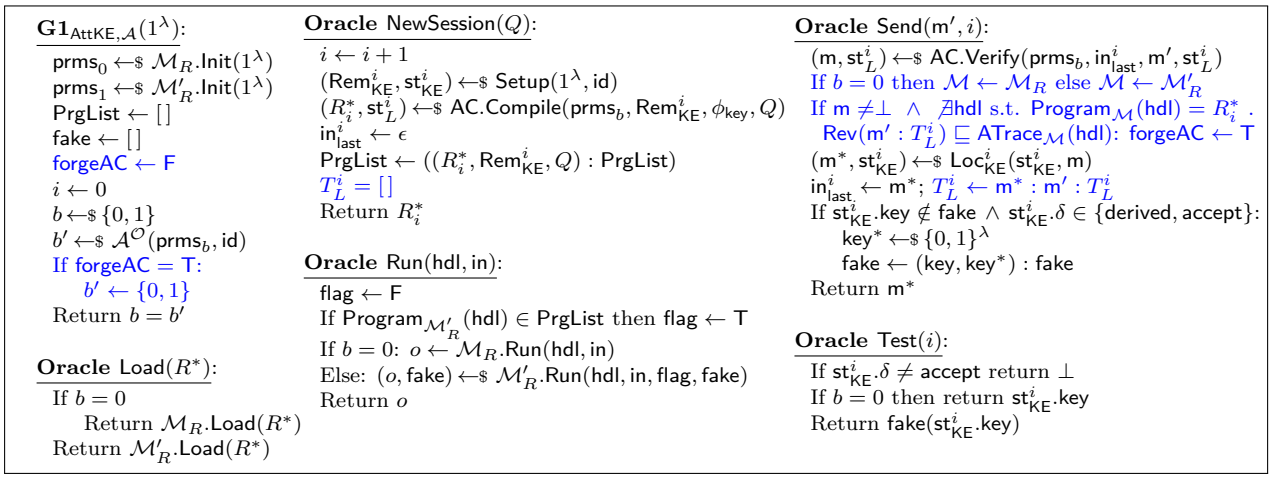

Figure 23: First hop of the utility proof.

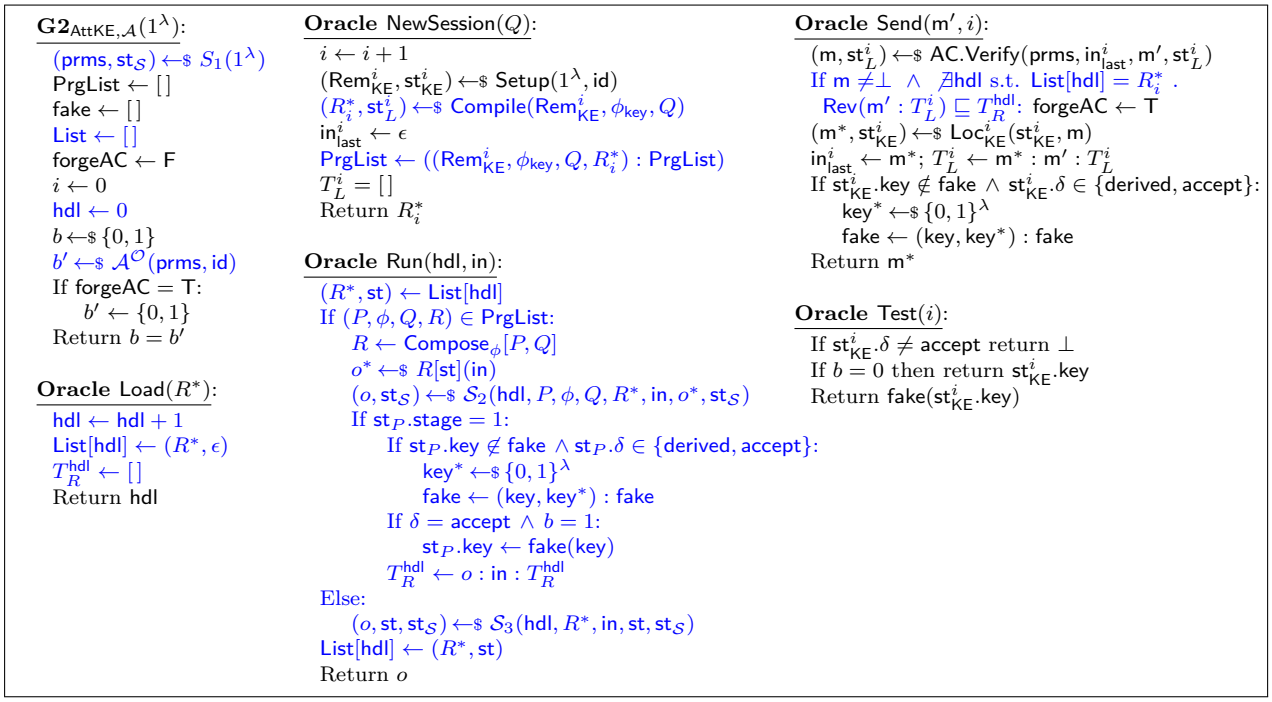

Figure 24: Second hop of the utility proof.

In the second game $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$, the adversary loses whenever a $\mathsf{forgeAC}$ event occurs. Intuitively, this event corresponds to the adversary producing an output that is successfully validated, but was not computed using Run, and hence constitutes a forgery with respect to AC. We establish that Rev reverses a list given as input[5]. Given that the two games are identical until this event occurs, we have that

$$\Pr[\,\mathsf{G0}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{forgeAC}\,].$$

[5]This is for handling a technicality, in which the trace of $\mathcal{M}$ and the constructed T^i_L are in reverse order.

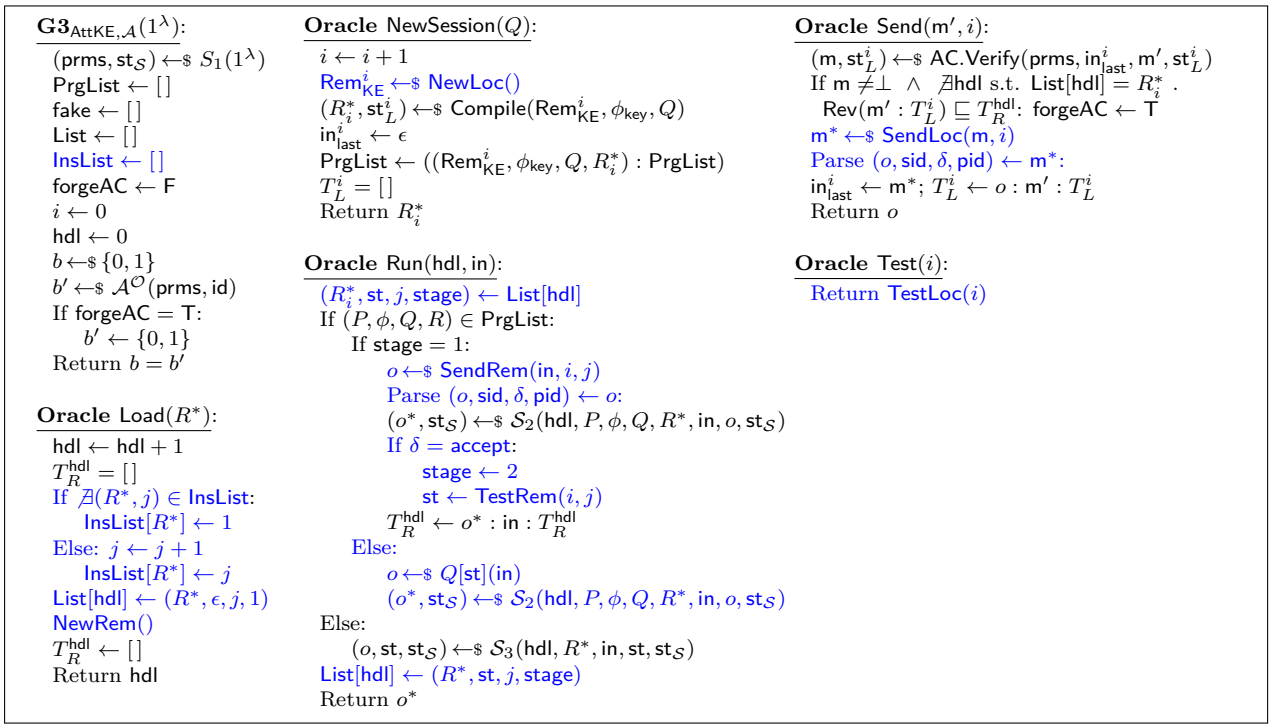

Figure 25: Third hop of the utility proof.

Let q be the maximum number of calls to $\mathsf{NewSession}$ allowed, and N the number of messages exchanged in $\mathsf{Rem}^i_{\mathsf{KE}}$. We upper bound the distance between these two games, by constructing an adversary $\mathcal{B}$ against the security of AC such that

$$\Pr[\,\mathsf{forgeAC}\,] \;\leq\; \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AC},\mathcal{B}}(\lambda) * \mathsf{q} * \lceil N/2 \rceil\,.$$

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$ as follows: at the beginning of the game, $\mathcal{B}$ has to try and guess which session will have a forged message, and which message of the protocol it will be. As such, it samples uniformly from $[1..\mathsf{q}]$ a session s and from $[1..\lceil N/2 \rceil]$ a message k. During $\mathsf{NewSession}$ such that $i = s$, $\mathcal{B}_1$ will output $(\mathsf{Rem}^i_{\mathsf{KE}}, \phi_{\mathsf{key}}, Q, k, \perp)$ to $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$, store the produced R^* and all hdl output by $\mathsf{Load}(R^*)$ from there on. Afterwards, all calls to $\mathcal{M}_R.\mathsf{Load}(R^*)$ and $\mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, m)$ or $\mathcal{M}'_R.\mathsf{Load}(R^*)$ and $\mathcal{M}'_R.\mathsf{Run}(\mathsf{hdl}, m, \mathsf{flag}, \mathsf{fake})$ with the same calls on $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$. Whenever $\mathsf{Send}(\mathsf{m}', i)$ is called, $i = s$ and $\mathsf{m} \neq \perp$, let n be the number of messages sent for session s:

- If $n = 0$, $\mathcal{B}_2$ outputs $(\perp, \mathsf{m}', \mathsf{m}^*)$.
- If $n > 0 \wedge n < k$, $\mathcal{B}_2$ outputs $(\mathsf{st}_\mathcal{B}, \mathsf{m}', \mathsf{m}^*)$

It remains to show that, when $\mathsf{forgeAC}$ is set, $\mathcal{B}$ wins $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$ with probability $1/(\mathsf{q} * \lceil N/2 \rceil)$.

When the game ends and $\mathsf{forgeAC} = \mathsf{T}$, we have that

$$\mathsf{m} \neq \perp\ \wedge\ \nexists \mathsf{hdl}\ \text{s.t.}\ \mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = R^*_i\ .\ (\mathsf{m}' : T^i_L) \sqsubseteq \mathsf{ATrace}_\mathcal{M}(\mathsf{hdl})$$

From the construction of Send, $\mathsf{m} \neq \perp$ means that this message has been successfully validated by $\mathsf{AC.Verify}$. Furthermore, this matches the verifications in $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$, considering that $\mathsf{in}^i_{\mathsf{last}} \leftarrow$

m^* in Send, so we know that all verifications that succeed in $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$ also do so in $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$. If $\mathsf{forgeAC}$ is set upon receiving message k, we reach the final check. All calls that produce handles such that $\mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}) = R_i^*$ are also performed in $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$, so it remains to show that $\mathsf{Rev}(\mathsf{m}' : T_L^i)$ matches T.

We know that $T \leftarrow \mathsf{ATrace}_{R[\mathsf{st};\mathsf{Coins}_{\mathcal{M}_R}(\mathsf{hdl}^*)]}(i_1, \ldots, i_n)$. From the construction of Send and the behavior of the adversary, we know that this is constructed with the outputs given by $\mathcal{B}_2$. We will show, inductively, why $\mathsf{Rev}(\mathsf{m}' : T_L^i)$ matches the k outputs of $\mathcal{B}_2$.

- Initially, given that $(\bot, \mathsf{m}'_0, \mathsf{m}^*_0) \leftarrow \mathcal{B}_2$:

$$\mathsf{Rev}(\mathsf{m}'_0 : T_L^i) = \mathsf{Rev}[\mathsf{m}'_0, \bot] = [\bot, \mathsf{m}'_0]$$

- For all subsequent messages up to k, $(\mathsf{m}^*_{n-1}, \mathsf{m}'_n, \mathsf{m}^*_n) \leftarrow \mathcal{B}_2$, so

$$\mathsf{Rev}(\mathsf{m}' : T_L^i) = \mathsf{Rev}[\mathsf{m}'_n, \mathsf{m}^*_{n-1}, \ldots, \mathsf{m}'_0, \bot] = [\bot, \mathsf{m}'_0, \ldots, \mathsf{m}^*_{n-1}, \mathsf{m}'_n]$$

As such, when $\mathsf{forgeAC}$ is set on session s, and in the k-th message input on Send, we have that

$$\nexists \mathsf{hdl} \text{ s.t. } \mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}) = R_i^* \ . \ (\mathsf{m}' : T_L^i) \sqsubseteq \mathsf{ATrace}_{\mathcal{M}}(\mathsf{hdl})$$

which results in a winning output for $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$ with probability $1/(\mathsf{q} * \lceil N/2 \rceil)$.

In the third game $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$, when the adversary loads and runs code, it is no longer interacting with machine $\mathcal{M}$, but rather with a simulator that is given the trace of a legitimate execution. Furthermore, $\mathcal{S}$ handles the different behavior of $\mathcal{M}_R$ and $\mathcal{M}'_R$ following the description in Section 7. Intuitively, this difference corresponds to the indistinguishable scenario presented in the minimum leakage game of Figure 6. Given the presented differences, we have that

$$\Pr[\,\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{Leak\text{-}Real}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{Leak\text{-}Ideal}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,].$$

We must now argue that the difference between the games is bound by the adversary's advantage of breaking minimal leakage. In both possibilities for the bit b, the transformation in Load is exactly the same as the one in the minimum leakage game. Furthermore, T_R^{hdl} will always correspond to the $\mathsf{ATrace}_{\mathcal{M}}(\mathsf{hdl})$. First, consider $b = 0$. The behavior of Run is exactly the same as the one in the minimum leakage game, modulo the generation of the fake key, which will not be taken into consideration since it is only used in Test when $b = 1$. Now consider $b = 1$. The behavior of Run is exactly the same as the one in the minimum leakage game, modulo generating and setting the fake key. However, these additional operations match the described behavior expected from $\mathcal{M}'_R$ for establishing the fake key.

As such, the advantage of the adversary in $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$ with respect to $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$ is limited by its advantage of breaking the minimum leakage.

Finally, in game $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$, we no longer run the key exchange part of the code, but rather make use of AttKE and its corresponding oracles $\{\mathsf{NewRem}, \mathsf{NewLoc}, \mathsf{SendRem}, \mathsf{SendLoc}, \mathsf{TestRem}, \mathsf{TestLoc}\}$. The intuition is that, at this fourth game, calls to the attested part of the protocol can be provided using a key exchange for attested computation AttKE. More formally, we want to show that

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

Whenever $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ creates a local or a remote session (respectively, NewSession or Load), it will initialize accordingly on the AttKE protocol. On the remote side, a new InsList will also keep track of how many remote sessions are created (just like oracle NewRem in AttKE). On the local side, algorithm Setup will be replaced by a call to NewLoc that by its construction will return the same value as in $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$. As such, it must now be argued that, every time the game is required to produce either an output from the local/remote machine or a key, the response given by the corresponding oracle is only distinguishable by an adversary that breaks AttKE security.

- For $\mathsf{stage} = 1 \wedge \delta \neq \mathsf{accept}$: Locally, SendLoc replaces calling $\mathsf{Loc}^i_{\mathsf{KE}}$, which by the oracle behavior implies the same result. Remotely, instead of producing output via $R[\mathsf{st}](\mathsf{in})$, we now directly execute oracle SendRem, which by construction holds and updates st in the same way as $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$.
- For $\mathsf{stage} = 1 \wedge \delta = \mathsf{accept}$: In $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$, SendLoc would replace the real key with a fake key according to decision bit b. In $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$, the same thing happens, but according to AttKE decision bit. Remotely, the behavior of $\mathsf{st} \leftarrow \mathsf{TestRem}$ also differs in the same manner.
- For $\mathsf{stage} = 2$: We know by $\mathsf{Compose}_\phi[P, Q]$, that the state st considered for Q only maintains the key. As such, from thereon, the only difference between $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$ and $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ is the key depending on the decision bit taken into consideration.

In this final game, the adversary's decision is based on a bit that is unrelated to any information obtainable via the oracles. Let

$$\mathsf{Adv}^{\mathsf{Leak}}_{\mathsf{AC},\mathcal{A}} = \Pr[\,\mathsf{Leak\text{-}Real}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{Leak\text{-}Ideal}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,]$$

we have that

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{Ut}}_{\mathsf{AttKE},\mathcal{A}} &= \Pr[\,\mathsf{G}_0^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)\,] - \Pr[\,\mathsf{G}_3^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)\,] \\
&= \sum_{i=0}^{3} \Pr[\,\mathsf{G}_i^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)\,] - \Pr[\,\mathsf{G}_{i+1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)] \\
&\leq \Pr[\mathsf{forgeAC}] + \mathsf{Adv}^{\mathsf{Leak}}_{\mathsf{AC},\mathcal{A}} + \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AttKE},\mathcal{A}} \\
&\leq \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AC},\mathcal{B}}(\lambda) * \mathsf{q} * \lceil N/2 \rceil + \mathsf{Adv}^{\mathsf{Leak}}_{\mathsf{AC},\mathcal{A}} + \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AttKE},\mathcal{A}}
\end{aligned}$$

and Theorem 3 follows.

□

D Proof of Theorem 4

In this section we provide the proof of security for the scheme defined in Section 8. In all the Section we assume that the encryption scheme (E, D, K) used is a secure authenticated encryption scheme. Therefore this encryption scheme satisfies both IND-CPA and INT-CTXT. In this section, let AttKE be the AttKE scheme used in the construction of our SOC scheme and AC be the AC scheme used.

D.1 Integrity

The proof consists of two game hops described in Figure 26. The first hop from G_0 to G_1 consists in using the utility of the key exchange to replace the shared key by a magically shared fresh key. The second game hop from G_1 to G_2 is simply using sequence numbers and integrity of the encryption scheme to ensure that the local and remote traces actually match. Let us now give the details of each game hop.

G_0 to G_1:

Let $\mathcal{A}$ be an adversary against G_0 or G_1, let us build an adversary $\mathcal{B}$ against $\mathsf{Att}_{\mathsf{AttKE}}$. The machine $\mathcal{B}$ simulates $\mathcal{A}$ giving $\mathsf{NewSession}(P)$ as input to $\mathcal{A}_2$ instead of $\mathsf{Compile}^{\mathrm{sec}}(\mathsf{prms}, P, \mathsf{id})$, answering the oracle calls as follows:

$\mathsf{BootStrap}(o)$**:** Return $\mathsf{Send}(o, 0)$

$\mathsf{Send}(o^*, i)$**:** $k \leftarrow \mathsf{Test}(0)$
$o, c \leftarrow D(o^*, k)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow E(\mathsf{count}_l \# i, k)$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

$\mathsf{Load}(R^*)$**:** Return $\mathsf{Load}(R^*)$

$\mathsf{Run}(\mathsf{hdl}, i)$**:** Return $\mathsf{Run}(\mathsf{hdl}, i)$

We now remark that if the bit b chosen in the utility game is 0, the game being played by $\mathcal{B}$ is exactly the integrity game. On the other hand, if the bit chosen is 1, either $\mathcal{B}$ violates entity authentication or $\mathcal{B}$ behaves as $\mathcal{A}$ playing against G_1. We conclude that

$$|G_0\text{-}\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - G_1\text{-}\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)| \leq \mathsf{Adv}^{\mathsf{utility}}_{\mathsf{AttKE},\mathcal{B}}(1^\lambda)$$

D.1.1 G_1 to G_2:

First note that the only difference between G_1 and G_2 occurs when the forge event is raised (the other differences are simple rewritings). Now let $\mathcal{A}$ be an adversary against G_2. We build an adversary $\mathcal{B}$ against the INT-CTXT game by simulating $\mathcal{A}$ playing G_2 using the encryption/decryption oracles provided by the INT-CTXT game. Note that the event forge is raised if and only if $\mathcal{B}$ wins the INT-CTXT game. Indeed, remarking that the counters are strictly increasing ensure that no encryption can be accepted twice as input to Run or Send. This entails the fact that if a message is accepted, it was the last message produced. We conclude that

$$|G_1\text{-}\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - G_2\text{-}\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)| \leq \mathsf{Adv}^{\text{INT-CTXT}}_{(E,D,K),\mathcal{B}}(1^\lambda)$$

It is now enough to remark that in G_2, the Run oracle and the Send oracle agree on inputs and outputs (unless forge is raised). As G_2 always returns true, we conclude

$$\mathsf{Adv}^{\mathsf{Int}}_{\mathsf{SOC}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{utility}}_{\mathsf{AttKE}}(1^\lambda) + \mathsf{Adv}^{\text{INT-CTXT}}_{(E,D,K)}(1^\lambda)$$

Game G_0-$\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
$\mathsf{prms} \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
Run $\mathcal{A}_2^{\mathsf{BootStrap},\mathsf{Send},\mathsf{Run},\mathsf{Load}}(st_\mathcal{A}, P^*)$
If $\nexists_{=1}\mathsf{hdl}$ such that
  $\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) = P^* \wedge$
  $\mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}_R(\mathsf{hdl})}) \neq []$
 Return F
$\mathsf{hdl} \leftarrow \mathsf{Program}^{-1}_{\mathcal{M}_R}(P^*)$
$T \leftarrow \mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}_R(\mathsf{hdl})})$
$T' \leftarrow \mathsf{tr}$
Return $\neg\Psi(T, T')$

Oracle BootStrap(i):
If $\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle Send($o*, i$):
$o, c \leftarrow D(o^*, \mathsf{st}_l.\mathsf{key})$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow E(\mathsf{count}_l \# i, \mathsf{st}_l.\mathsf{key})$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle Load(R^*):
$\mathsf{hdl} \leftarrow \mathcal{M}_R.\mathsf{Load}(R^*)$
Return hdl

Oracle Run(hdl, i):
$o \leftarrow\$ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i)$
Return o

Game G_1-$\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
$k \leftarrow\$ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{prms} \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
Run $\mathcal{A}_2^{\mathsf{BootStrap},\mathsf{Send},\mathsf{Run},\mathsf{Load}}(st_\mathcal{A}, P^*)$
If $\nexists_{=1}\mathsf{hdl}$ such that
  $\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) = P^* \wedge$
  $\mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}_R(\mathsf{hdl})}) \neq []$
 Return F
$\mathsf{hdl} \leftarrow \mathsf{Program}^{-1}_{\mathcal{M}_R}(P^*)$
$T \leftarrow \mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}_R(\mathsf{hdl})})$
$T' \leftarrow \mathsf{tr}$
Return $\neg\Psi(T, T')$

Oracle BootStrap(o):
If $\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle Send($o*, i$):
If $\neg\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$o, c \leftarrow D(o^*, k)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow E(\mathsf{count}_l \# i, k)$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle Load(R^*):
$\mathsf{hdl} \leftarrow \mathcal{M}_R.\mathsf{Load}(R^*)$
Return hdl

Oracle Run(hdl, i^*):
If $\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) \neq P^*$
  $o^* \leftarrow\$ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\neg\mathsf{st}_{\mathcal{M}_R}(\mathsf{hdl}).\mathsf{accept}$
  $o \leftarrow\$ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\exists \mathsf{hdl}_2 \neq \mathsf{hdl}.\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) = P^*$
    $\wedge \mathsf{st}_{\mathcal{M}_R}(\mathsf{hdl}).\mathsf{accept}$
  raise twoPartners
$i \# c \leftarrow D(i^*, k)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
If $c \neq \mathsf{count}_R$ Return $\perp$
$o \leftarrow P[\mathsf{st}_P](i)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
Return $E(\mathsf{count} \# o, k)$

Game G_2-$\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
$k \leftarrow\$ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{tr}_R \leftarrow []$
$\mathsf{prms} \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
Run $\mathcal{A}_2^{\mathsf{BootStrap},\mathsf{Send},\mathsf{Run},\mathsf{Load}}(st_\mathcal{A}, P^*)$
If $\nexists_{=1}\mathsf{hdl}$ such that
  $\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) = P^* \wedge$
  $\mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}_R(\mathsf{hdl})}) \neq []$
 Return F
$\mathsf{hdl} \leftarrow \mathsf{Program}^{-1}_{\mathcal{M}_R}(P^*)$
$T \leftarrow \mathsf{tr}_R$
$T' \leftarrow \mathsf{tr}$
Return $\neg\Psi(T, T')$

Oracle BootStrap(o):
If $\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle Send($o*, i$):
If $\neg\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$o, c \leftarrow D(o^*, k)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
If $o \neq \mathsf{last}_o$ Raise forge
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow E(\mathsf{count}_l \# i, k)$
$\mathsf{last}_i \leftarrow i$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle Load(R^*):
$\mathsf{hdl} \leftarrow \mathcal{M}_R.\mathsf{Load}(R^*)$
Return hdl

Oracle Run(hdl, i^*):
If $\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) \neq P^*$
  $o^* \leftarrow\$ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\neg\mathsf{st}_{\mathcal{M}_R}(\mathsf{hdl}).\mathsf{accept}$
  $o \leftarrow\$ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\exists \mathsf{hdl}_2 \neq \mathsf{hdl}.\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) = P^*$
    $\wedge \mathsf{st}_{\mathcal{M}_R}(\mathsf{hdl}).\mathsf{accept}$
  raise twoPartners
$i \# c \leftarrow D(i^*, k)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
If $c \neq \mathsf{count}_R$ Return $\perp$
If $i \neq \mathsf{last}_i$ Raise forge
$o \leftarrow P[\mathsf{st}_P](i)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
$\mathsf{last}_o \leftarrow o$
$\mathsf{tr}_R \leftarrow o : i : \mathsf{tr}_R$
Return $E(\mathsf{count} \# o, k)$

Figure 26: Game hops for integrity of the SOC scheme

D.2 Privacy

The proof of consists, as previously in replacing the derived key by a magically shared key. The second game hop, as in the integrity property, makes sure that the inputs received on both sides coincide. The game G_2 obtained reduces then quite simply to IND-CPA of the encryption scheme. The game hop is presented in Figure 27. The reduction from G_0 to G_1

is exactly the same as the one in the integrity proof, we do not detail it further. We get as previously (for some PPT $\mathcal{B}$)

$$\left|G_0\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - G_1\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)\right| \leq \mathsf{Adv}^{\mathsf{utility}}_{\mathsf{AttKE},\mathcal{B}}(1^\lambda)$$

G_1 to G_2:

Let $\mathcal{A}$ be an adversary against G_1 or G_2. Let us build an adversary $\mathcal{B}$ against the INT-CTXT game for (E, D, K). The machine $\mathcal{B}$ simulates $\mathcal{A}$ playing the game G_1, with the exception of not drawing k and using the encryption and decryption oracle of the INT-CTXT game. Remark that G_1 and G_2 behave differently only if $\mathcal{A}$ is able to submit a forged encryption to either the send or the run oracle. Indeed, if the encryption sunmitted is neither the last produced encryption nor a forgery, the sequence number makes sure that the corresponding oracle return $\perp$. We get

$$\left|G_1\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - G_2\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)\right| \leq \mathsf{Adv}^{\text{INT-CTXT}}_{(E,D,K),\mathcal{B}}(1^\lambda)$$

Reducing G_2 to IND-CPA:

The key point in the proof is reducing G_2 to the security of the authenticated encryption scheme. Let us call Chal the IND-CPA challenge oracle and Enc the encryption oracle in the IND-CPA game for $(E, D.K)$. Let $\mathcal{A}$ be an adversary against G_2-Priv. We build the following adversary $\mathcal{B}$ against the IND-CPA game for (E, D, K). The machine $\mathcal{B}$ simulates the game G_2, without drawing the key k or the bit b and using the IND-CPA oracles to perform encryptions. In the Send oracle, instead of computing $E(\mathsf{count}_l\#i_b, k)$, $\mathcal{B}$ calls $\mathsf{Chal}(\mathsf{count}_l\#i_0, \mathsf{count}_l\#i_1)$. Similarly, in the Run oracle, instead of computing $E(\mathsf{count}\#o_b, k)$, $\mathcal{B}$ calls $\mathsf{Chal}(\mathsf{count}\#o_0, \mathsf{count}\#o_1)$. All calls are well formed as at each call of Send, the length of the two inputs is required to be identical and, in the Run oracle, P is assumed length-uniform. It is now enough to remark that $\mathcal{B}$ wins the IND-CPA game if and only if $\mathcal{A}$ wins G_0 to conclude

$$\left|G_2\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - \frac{1}{2}\right| \leq \mathsf{Adv}^{\text{IND-CPA}}_{(E,D,K),\mathcal{B}}(1^\lambda)$$

From this result, we can sum everything up and obtain the advantage of an adversary against the privacy game:

$$\mathsf{Adv}^{\mathsf{Priv}}_{\mathsf{SOC}}(1^\lambda) \leq \mathsf{Adv}^{\text{IND-CPA}}_{(E,D,K),\mathcal{B}}(1^\lambda) + \mathsf{Adv}^{\text{INT-CTXT}}_{(E,D,K),\mathcal{B}}(1^\lambda) + \mathsf{Adv}^{\mathsf{utility}}_{\mathsf{AttKE},\mathcal{B}}(1^\lambda)$$

Game G_0-$\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
$k \leftarrow\$\ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{prms} \leftarrow\$\ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}}(st_\mathcal{A}, P^*)$
Return $b = b'$

Oracle $\mathsf{BootStrap}(o)$:
If $\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle $\mathsf{Send}_b(o*, i_0, i_1)$:
If $|i_0| \neq |i_1|$ Return $\perp$
$o, c \leftarrow D(o^*, \mathsf{st}_l.\mathsf{key})$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow E(\mathsf{count}_l \# i_b, \mathsf{st}_l.\mathsf{key})$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle $\mathsf{Load}(R^*)$:
$\mathsf{hdl} \leftarrow \mathcal{M}_R.\mathsf{Load}(R^*)$
Return hdl

Oracle $\mathsf{Run}(\mathsf{hdl}, i^*)$:
$o^* \leftarrow\$\ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i^*)$
Return o^*

Game G_1-$\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
$k \leftarrow\$\ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{prms} \leftarrow\$\ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}}(st_\mathcal{A}, P^*)$
Return $b = b'$

Oracle $\mathsf{BootStrap}(o)$:
If $\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle $\mathsf{Send}_b(o*, i_0, i_1)$:
If $|i_0| \neq |i_1|$ Return $\perp$
If $\neg\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$o, c \leftarrow D(o^*, k)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow E(\mathsf{count}_l \# i_b, k)$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle $\mathsf{Load}(R^*)$:
$\mathsf{hdl} \leftarrow \mathcal{M}_R.\mathsf{Load}(R^*)$
Return hdl

Oracle $\mathsf{Run}(\mathsf{hdl}, i^*)$:
If $\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) \neq P^*$
  $o^* \leftarrow\$\ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\neg\mathsf{st}_{\mathcal{M}_R}(\mathsf{hdl}).\mathsf{accept}$
  $o \leftarrow\$\ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\exists \mathsf{hdl}_2 \neq \mathsf{hdl}.\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) = P^*$
    $\wedge \mathsf{st}_{\mathcal{M}_R}(\mathsf{hdl}).\mathsf{accept}$
  raise twoPartners
$i\#c \leftarrow D(i^*, k)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
If $c \neq \mathsf{count}_R$ Return $\perp$
$o \leftarrow P[\mathsf{st}_P](i)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
Return $E(\mathsf{count} \# o, k)$

Game G_2-$\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
$k \leftarrow\$\ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{prms} \leftarrow\$\ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}}(st_\mathcal{A}, P^*)$
Return $b = b'$

Oracle $\mathsf{BootStrap}(o)$:
If $\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle $\mathsf{Send}_b(o*, i_0, i_1)$:
If $|i_0| \neq |i_1|$ Return $\perp$
If $o^* \neq \mathsf{last}_{o^*}$ Return $\perp$
$i* \leftarrow E(\mathsf{count}_l \# i_b, k)$
$\mathsf{last}_{i_0} \leftarrow i_0$
$\mathsf{last}_{i_0} \leftarrow i_1$
$\mathsf{last}_{i^*} \leftarrow i^*$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle $\mathsf{Load}(R^*)$:
$\mathsf{hdl} \leftarrow \mathcal{M}_R.\mathsf{Load}(R^*)$
Return hdl

Oracle $\mathsf{Run}_b(\mathsf{hdl}, i^*)$:
If $\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) \neq P^*$
  $o^* \leftarrow\$\ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i)$
  Return o^*
If $\neg\mathsf{st}_{\mathcal{M}_R}(\mathsf{hdl}).\mathsf{accept}$
  $o \leftarrow\$\ \mathcal{M}_R.\mathsf{Run}(\mathsf{hdl}, i)$
  Return o^*
If $\exists \mathsf{hdl}_2 \neq \mathsf{hdl}.\mathsf{Program}_{\mathcal{M}_R}(\mathsf{hdl}) = P^*$
    $\wedge \mathsf{st}_{\mathcal{M}_R}(\mathsf{hdl}).\mathsf{accept}$
  raise twoPartners
If $i \neq \mathsf{last}_{i^*}$ Return $\perp$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
$o_0 \leftarrow P[\mathsf{st}_P^0](\mathsf{last}_{i_0})$
$o_1 \leftarrow P[\mathsf{st}_P^1](\mathsf{last}_{i_1})$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
$o^* \leftarrow E(\mathsf{count} \# o_b, k)$
$\mathsf{last}_{o^*} \leftarrow o^*$
Return o^*

Figure 27: Game hops in the privacy proof