π_{RA}: A π-calculus for Verifying Protocols that Use Remote Attestation

Emiel Lanckriet
KU Leuven
Leuven, Belgium
emiel.lanckriet@kuleuven.be

Matteo Busi
Ca' Foscari University of Venice
Venice, Italy
matteo.busi@unive.it

Dominique Devriese
KU Leuven
Leuven, Belgium
dominique.devriese@kuleuven.be

2023 IEEE 36th Computer Security Foundations Symposium (CSF) | 979-8-3503-2192-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/CSF57540.2023.00019

***Abstract*—Remote attestation (RA) is a primitive that allows the authentication of software components on untrusted systems by relying on a root of trust. Network protocols can use the primitive to establish trust in remote software components they communicate with. As such, RA can be regarded as a first-class security primitive like (a)symmetric encryption, message authentication, etc. However, current formal models of RA do not allow analysing protocols that use the primitive without tying them to specific platforms, low-level languages, memory protection models, or implementation details.**

In this paper, we propose and demonstrate a new model, called π_{RA}, that supports RA at a high level of abstraction by treating it as a cryptographic primitive in a variant of the applied π-calculus. To demonstrate the use of π_{RA}, we use it to formalise and analyse the security of MAGE, an SGX-based framework that allows mutual attestation of multiple enclaves. The protocol is formalised in the form of a compiler that implements actor-based communication primitives in a source language (π_{Actor}) in terms of remote attestation primitives in π_{RA}. Our security analysis uncovers a caveat in the security of MAGE that was left unmentioned in the original paper.

***Index Terms*—remote attestation, secure compilation, π-calculus**

I. Introduction

Remote attestation (RA) is a security primitive that allows authenticating software components on untrusted systems by relying on a root of trust. This primitive exploits the root of trust (a trusted hardware, firmware, or software entity) to produce an attest for a given (software) component. An attest is essentially an authenticated token from the root of trust and vouches for the identity of its corresponding component. The identity of a component is typically defined in terms of the component's executable code (e.g., it may include a hash of the component's machine code).

Once attested, a component can use its attest to prove its identity to locally and remotely located third parties, allowing them to establish trust in the code that the component is executing. Crucially, this process is trustworthy only if the code of the component has not been tampered with after the attestation process. Indeed, if an attested piece of code is changed by a third party, its identity may change, making the attest outdated. For this reason, attestation primitives are often tied to isolation primitives (e.g., enclaves, virtual machines, etc.) that separate the attested code from other, untrusted code running on the same system. We will refer to these already attested regions as attested components.

Support for RA has grown substantially in the last decade with several mature implementations for full-fledged processors [1], [2], embedded processors [3], [4], and separate chips supporting the processor [5]. Not all implementations of RA take the form of low-level APIs supported by hardware primitives. Examples include schemes for software-based RA on resource-constrained devices [6] and RA APIs in high-level languages, such as secureworker in JavaScript [7].

Software developers can reason about RA similarly to other cryptographic primitives like (a)symmetric encryption, cryptographic signatures, or secure hashes. Defining and reasoning about protocols built on such cryptographic primitives is best done in a setting where certain details are abstracted away, like the choice of primitive instantiation and their security parameters, or the distribution and communication model (e.g., communicate locally over IPC primitives or remotely over a wireless network). For protocols built on RA, one should abstract away the choice of RA implementation, the access model used to isolate attested code from untrusted code, or the fact that attestation primitives are usually offered at assembly level on von Neumann machines.

To analyse protocols based on security primitives, such as RA or (a)symmetric encryption, probabilistic and symbolic methods are used. Although probabilistic methods are more accurate, symbolic methods are often simpler as they assume that primitives provide idealised guarantees. One popular framework for constructing symbolic models of systems supporting cryptographic primitives is the applied π-calculus [8], an extension of the π-calculus that is well suited for modelling communication and distributed systems. The applied π-calculus adds support for an extensible equational theory, which allows modelling idealised cryptographic primitives. For example, symmetric encryption can be modelled as a pair of functions $\mathit{encrypt}(M, k)$ and $\mathit{decrypt}(M, k)$, such that $\mathit{decrypt}(\mathit{encrypt}(M, k), k) = M$. This model is idealised because brute forcing and lucky guesses are not just statistically unlikely but simply impossible. Unfortunately, the applied π-calculus cannot directly model remote attestation primitives

This work was partially supported by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity, by the Air Force Office of Scientific Research under award number FA9550-21-1-0054 and by the Research Foundation - Flanders (Research Project G030320N).

DOI 10.1109/CSF57540.2023.00019

since RA uses cryptographic values (attests) that depend on the executable code of software components (see Section III).

This paper proposes a formal model for reasoning about RA-based protocols at a high level of abstraction. The model is a variant of the applied π-calculus, called $\pi_{\mathbf{RA}}$, where attestation is handled as a cryptographic primitive. The model abstracts away from details like the encapsulation model, components' low-level code, and the protocol used for attestation. Note that the primary goal of $\pi_{\mathbf{RA}}$ is to provide a tool for reasoning about protocols that build upon the mechanisms of RA, not for modelling RA implementations (see Section VI).

To demonstrate the use of $\pi_{\mathbf{RA}}$ for reasoning about protocols, we study the MAGE protocol [9]. This protocol uses RA primitives for mutually authenticating a group of components (the problem and solution are explained in more detail in Section II-c). We model the action of adding MAGE's mutual attestation features to any protocol as a compiler that implements authenticated actor communication primitives securely in terms of $\pi_{\mathbf{RA}}$'s RA primitives. We prove a secure compilation result establishing that the translation accurately preserves source language guarantees. Our security analysis formalises the precise guarantees that MAGE offers, clarifying a security caveat that went unmentioned in the original paper.

This paper makes the following contributions:

- $\pi_{\mathbf{RA}}$: an extension of the applied-π calculus that provides RA as a set of hardware-independent primitives and enables abstract reasoning about protocols using RA.
- A formal security analysis of the MAGE protocol, consisting of (1) a source language π_{Actor} with secure actor-based communication primitives, (2) a formal compiler from π_{Actor} to $\pi_{\mathbf{RA}}$ that defines the MAGE protocol in a more abstract setting and (3) a secure compilation result from π_{Actor} to $\pi_{\mathbf{RA}}$[1] establishing the security of MAGE.
- Our analysis clarifies a security caveat for using MAGE that was left unmentioned in the original paper.

Structure of the paper: Section II introduces a few preliminaries after which Section III introduces $\pi_{\mathbf{RA}}$. Section IV defines π_{Actor}, models MAGE as a compiler from π_{Actor} to $\pi_{\mathbf{RA}}$ and discusses our formal statement of its security. Section V reports on our proof of the security theorems. Sections VI and VII overview-related literature and conclude.

II. Background

a) Remote attestation: As mentioned in the introduction, remote attestation is a security primitive that allows the authentication of software components on untrusted systems by relying on a root of trust. The primitive can be implemented using a static (S-RTM) or dynamic root of trust for measurement (D-RTM). According to the Trusted Computing Group [11], an S-RTM is a "chain of measurements of platform state that begin at Host Platform Reset (e.g., power-on or system restart) (...). The measurement chain continues with components measuring subsequent components and configuration data before passing control to them." This results in a large trusted computing base including the whole OS, hence one OS bug can break security. S-RTM is notably used in the Trusted Platform Module (TPM) [5]. A D-RTM [11] allows the measurement chain to start at any time so that the entire OS does not need to be trusted. Once an attest is generated with a D-RTM, the attested code can only be changed by itself, which prevents untrusted components from tampering with attested ones. Systems using a D-RTM include Intel SGX [2], MIT Sanctum [1], Sancus [3] and TrustLite [4].

The cryptography used to produce attests varies from system to system. Some systems use relatively simple methods based on symmetric encryption and pre-shared keys with a trusted third party [3]. Others use asymmetric encryption or even group cryptographic signatures to obviate the trusted third party or obtain additional anonymity properties. We regard these choices as orthogonal and will treat them abstractly in our model.

b) The applied π-calculus: In this section, we only present the parts of the applied π-calculus [8] that are featured in this paper. In particular, we omit communication primitives and active substitutions. Note that communication primitives are required to make concurrency useful, but both $\pi_{\mathbf{RA}}$ (cf. Section III) and π_{Actor} (cf. Section IV) will add their own.

The terms in the applied π-calculus are either names of channels, variables, or function applications on terms:

$L, M, N, T, U, V ::=$		terms
	$a, b, c, \ldots, k, \ldots, m, n, \ldots, s$	names
	x, y, z	variables
	$f(M_1, \ldots, M_l)$	function applications

All functions f are denoted in the signature Σ (not to be confused with the cryptographic signature). The signature Σ is a finite set of function symbols, along with equality rules, such as $f(a, b) = f(b, a)$ which denotes commutativity. If function applications cannot be proven equal according to these equality rules, then they are not equal in this framework.

The processes have the following syntax:

$P ::=$		processes
	$\mathbf{0}$	inactive processes
	$P_1 \mid P_2$	parallel composition
	$!\,P$	replication
	$P + Q$	non-deterministic choice
	$new\ a\ P$	restriction
	$if\ M = N\ then\ P\ else\ Q$	conditional
	$\mathrm{print(M).P}$	outputs M
	$\mathrm{userInput(z).P}$	user input in variable z

The restriction $new\ a\ P$ makes available a fresh (i.e., distinct from all others) name as variable a in P. The constructs $print(M)$ and $userInput(z)$ are shown in a different font because they are not in the applied π-calculus but added by

[1] To ease reading, we typeset elements of source languages, such as π_{Actor}, in blue, sans − serif, target elements, such as $\pi_{\mathbf{RA}}$, in **red, bold** and common ones in black [10].

us to more easily reason about traces in the proofs.[2] The construct $P_1 \mid P_2$ denotes the parallel composition of two programs, modelling concurrency. The replication $!\,P$ allows us to create non-terminating processes, e.g., recursion can be emulated using replication and communication between parallel programs.

Programs are equipped with a structural congruence relation, relating programs that should be treated as identical:

1) Programs that only differ up to a change of bound names are structurally congruent
2) Programs that only differ up to a reordering of terms in a summation are structurally congruent
3) $P \mid \mathbf{0} \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
4) $new\, u\, (P \mid Q) \equiv P \mid new\, u\, Q$ if $u \notin fv(P) \cup fn(P)$, $new\, u\, \mathbf{0} \equiv \mathbf{0}$, $new\, u\, (new\, v\, A) = new\, v\, (new\, u\, A)$
5) $!\,P \equiv P \mid !\,P$.

Here, $P \equiv Q$ means P and Q are structurally congruent and $fv(P)$ and $fn(P)$ denote the free variables and free names of P.

Figure 1 defines some evaluation rules where ground term denotes a term without free variables. The semantics of non-deterministic choice, parallel composition, if-then-else, print and userInput are given by rules TAU, PAR, THEN, ELSE, PRINT and USERINPUT. Structurally congruent programs behave the same (rule STRUCT) and rule RES defines that programs under a $new\, a$ evaluate as without the $new\, a$, as long as a is not leaked in inputs or outputs.[3] Note that choosing Q in $P + Q$ can be done by using TAU and STRUCT.

c) MAGE, an example protocol using remote attestation: The MAGE protocol [9] aims to solve the circularity problem in mutual attestation. This problem is depicted in Figure 2. If programs A and B want to authenticate each other using RA, then B must include the hash of A to verify A's identity and A similarly should contain the hash of B. Thus, the hash of A indirectly depends on itself and its definition is circular. MAGE breaks this loop by preprocessing some information as shown in Figure 3. The protocol starts by calculating the hashes of the code of each of the programs without the hashes of other programs, see the magenta box around the code of B. These hashes without other hashes are all calculated and compiled in a table, called idT. This table is concatenated with the program as depicted in Figure 3. If program A wants to verify the identity of program B, it reads the corresponding hash from the table. This is not yet the correct hash for B, as it does not contain B's copy of the table of hashes. However, A can derive the correct hash of B from the preprocessed hash of B and A's copy of the table.

For this derivation, MAGE relies on the special form of the hash function used on Intel SGX. Essentially, the hash of $A \| idT$ (idT concatenated to A) can be computed using the hash of A and the contents of idT (for block-aligned A and idT).

Interestingly, Zheng and Arden [12] proposed a different but very similar scheme to resolve the circularity in what they called the Decent Application Platform. They also use a preprocessed table, but they do not exploit the special form of the hash function and instead let enclaves send the table in use and check whether they are using the same table or not. Orthogonally to this, they also implement delegation of attestation, dynamic verifiers and revokers.

d) Fully abstract compilation and robust (relational) hyperproperty preservation: For our analysis, we formalise the intended security property of MAGE as a secure compilation result. The idea is that the security of a compiler implementing a protocol (e.g. MAGE) can be regarded as a security property of the protocol. It establishes that the protocol enforces certain guarantees, namely, the ones that hold in the source language. Essentially, applications that are securely implemented in terms of abstract source language primitives against a source attacker model, remain secure when the compiler implements the application using the protocol in terms of target language primitives, against the target language's attacker model. This is similar to real/ideal style proofs from cryptography (cf. [13] and [14]). For example, a compiler representing MAGE is a compilation from a language that takes mutual attestation for granted (ideal world) to $\pi_{\mathbf{RA}}$ (real world). If this compiler is subsequently shown to be secure, then compiling a program/protocol with this compiler corresponds to securely adding MAGE's mutual attestation features to that program/protocol. This also means that the secure compilation result proves security under composition. Also, note that the choice of source language (ideal world) is just as much a part of the security property as the secure compiler, because if the studied primitive is not trivially safe in the source language, then the security of the studied protocol is unclear.

Several secure compilation criteria exist in the literature, each with its advantages and disadvantages. Two prominent and fairly strict criteria of secure compilation are fully abstract compilation (FAC) [15], [16] and robust relational hyperproperty preservation (RrHP) [17].

To define these criteria, we must first define the behaviour of a program and the concept of contextual equivalence. We define the behaviour of a program as the set of all possible traces with which the program can evaluate:

$$behav(P) = \{t \mid P \rightsquigarrow t\}$$

where $P \rightsquigarrow t$ means that P can evaluate to some program P' with trace t. We define contextual equivalence in terms of this trace semantics (easing discussing RrHP and FAC together):

$$P \simeq_{ctx} Q \iff \forall C : behav(C[P]) = behav(C[Q])$$

The context C represents a (potentially adversarial) client and $C[P]$ means that program P is linked with C. Thus, contextual equivalence $P \simeq_{ctx} Q$ means that P and Q behave the same in every context or, in other words, no context can distinguish

[2]Alternatively, we could have used I/O on special-purpose channels, but separate primitives allow us to more easily distinguish internal I/O (with other processes in the system) from external I/O (with the outside world).

[3]Note that a can be renamed, so leaking a would result in undefined behaviour.

$$\textsc{Par}: \frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q} \qquad \textsc{Res}: \frac{P \xrightarrow{a} P'}{new\, x\, P \xrightarrow{a} new\, x\, P'} \text{if } x \notin a \qquad \textsc{Struct}: \frac{Q \equiv P \qquad P \xrightarrow{a} P' \qquad P' \equiv Q'}{Q \xrightarrow{a} Q'}$$

$$\textsc{Print}: print(M).P \xrightarrow{p(M)} P \qquad \textsc{Tau}: P + Q \to P \qquad \textsc{Then}: if\ M = M\ then\ P\ else\ Q \to P$$

$$\textsc{UserInput}: userInput(x).P \xrightarrow{ui(a)} P\left\{{}^{a}/_{x}\right\}$$

$$\textsc{Else}: if\ M = N\ then\ P\ else\ Q \to Q \qquad \text{for } M \text{ and } N \text{ ground and } \Sigma \nvdash M = N$$

Fig. 1: The evaluation rules of the applied π-calculus without communication

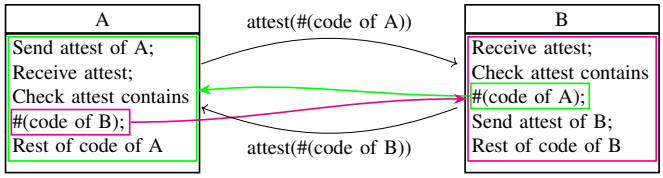

Fig. 2: Illustration of the circularity problem

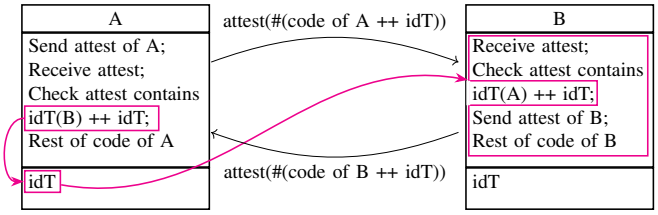

Fig. 3: Illustration of MAGE

them. If there are different secret values embedded in the source code of P and Q, then contextual equivalence implies confidentiality of the secret, as no context C can distinguish the two programs. We will often refer to contexts as attackers instead.

We can now define FAC for a compiler $[\![\cdot]\!]$ as follows:

$$\forall \mathsf{P}, \mathsf{Q} : \mathsf{P} \simeq_{\mathsf{ctx}} \mathsf{Q} \iff [\![\mathsf{P}]\!] \simeq_{\mathbf{ctx}} [\![\mathsf{Q}]\!]$$

This means that if the two source programs P and Q are indistinguishable to source-language contexts, then their compilations are also indistinguishable to target-language contexts. For example, specifically in the case of secrecy (as explained briefly above), FAC shows that if the source program protects a secret embedded in its code from an attacker, then its compilation will as well.

Next, RrHP [17] is defined as follows:

$$\forall R \in 2^{Behavs^{\omega}} : \forall \mathsf{P}_1, \ldots, \mathsf{P}_k, \ldots :$$
$$(\forall \mathsf{C_S} : (behav(\mathsf{C_S}[\mathsf{P}_1]), \ldots, behav(\mathsf{C_S}[\mathsf{P}_k]), \ldots) \in R) \implies$$
$$(\forall \mathbf{C_T} : (behav(\mathbf{C_T}[[\![\mathsf{P}_1]\!]]), \ldots, behav(\mathbf{C_T}[[\![\mathsf{P}_k]\!]]), \ldots) \in R)\,.$$

The set $R \in 2^{Behavs^{\omega}}$ is a relational hyperproperty, which is a relation between the behaviours of several programs. For example, contextual equivalence is a relational hyperproperty between two programs. RrHP states that the compiler preserves relational hyperproperties, such as R, meaning that if a set of programs satisfies the relation R then the set of their compilations also satisfies R. RrHP comes with an equivalent "property-free" characterisation called RrHC:

$$\forall \mathbf{C_T} : \exists \mathsf{C_S} : \forall \mathsf{P} : behav(\mathbf{C_T}[[\![\mathsf{P}]\!]]) = behav(\mathsf{C_S}[\mathsf{P}])$$

where $\mathbf{C_T}$ is a target context. In this paper, we will only use (a variant of) RrHC. An interpretation of RrHC is that a compiled program cannot be coerced to do more under the influence of a target attacker than the source program could be coerced to do under the influence of a source attacker. Contrary to other properties like the non-relational RHC, which we do not discuss in detail, RrHC requires that there exists a single context $\mathsf{C_S}$ that can accurately simulate the interaction of $\mathbf{C_T}$ with arbitrary programs P. The differences between FAC and RrHC are subtle and discussed elsewhere in more detail [17], but RrHC is more general: it does not only protect secrets embedded in a program's source code, but also implies the preservation of other properties than indistinguishability.

Essentially, we chose these criteria because FAC is a well-established criterion and RrHP is simply the strongest secure compilation criterion proposed to date.

Note that the meaning of these definitions can vary greatly based on what is considered a valid context, i.e., the set of valid contexts should correspond to the chosen attacker model. For example, in Section V we will carefully choose the right contexts to correspond to the power that MAGE allows attackers to have. The interested reader can explore these concepts more in-depth in the survey by Patrignani, Ahmed and Clarke [16] and the paper by Abate et al. [17].

III. The Model of Remote Attestation

In this section, we present our model of RA, called $\pi_{\mathbf{RA}}$, and motivate our choices. RA is not a traditional cryptographic primitive, as it is usually supported by (a combination of) hardware, software, and cryptography. Moreover, traditionally, whether a component can obtain a secret value depends only on whether it has the right inputs, while RA assumes a form of secret values (attests) that can be obtained only by specific components. However, in this paper, we choose to study RA as if it were a cryptographic primitive, and we model it in a variant of the applied π-calculus. This calculus' focus on communication allows modelling representative RA protocols and its equational theory allows guarantees to be

easily idealised symbolically. RA requires some additions to the applied π-calculus, but we restrict them to a minimum.

A. The π_{RA}-calculus

In this section, we discuss the terms, constructs, and evaluation rules of π_{RA}. First, we extend the signature with an irreversible function $attest(\cdot)$, modelling attests. Because one should not be able to deconstruct an attest, there are no rules for that in the equational theory and $attest(\mathbf{h}) = attest(\mathbf{h}')$ only holds if $\mathbf{h}' = \mathbf{h}$ holds. The cryptographic hash is also added to the signature and is denoted by $\#$. It maps a program or term to a term and the equational theory is again simple, with only hashes of the same program (up to the renaming of bound variables) being equal. When a hash of a program is taken, the program is viewed as just a string without operational relevance, hence only the syntax is relevant.

However, to model that attests can only be obtained by the program having that hash, we need to do more than just extending the signature. We assume that attests cannot appear as terms initially and can only be generated by a specific program construct, $getAttest(\mathbf{x}, \mathbf{d}, \mathbf{D}).\mathbf{P}$, that can read the whole program, $\mathbf{P}$, similar to the trusted processor reading the whole program. Furthermore, we add some constructs that make it easier to communicate using attests as authentication. Specifically, the following constructs are added:

$$\begin{aligned} \mathbf{P} ::= \cdots \mid\ & getAttest(\mathbf{x}, \mathbf{d}, \mathbf{D}).\mathbf{P} && \text{attested process} \\ & \overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M}, \mathbf{a}, \mathbf{h}\rangle.\mathbf{P} && \text{authenticated send} \\ & \mathbf{N}^{\mathbf{auth}}(\mathbf{y}, \mathbf{h}, \mathbf{a}).\mathbf{P} && \text{authenticated receive.} \end{aligned}$$

The construct $getAttest(\mathbf{x}, \mathbf{d}, \mathbf{D}).\mathbf{P}$ (where $\mathbf{x}$ and $\mathbf{d}$ are free in $\mathbf{P}$) evaluates to $\mathbf{P}\left\{{}^{attest(\#_{\mathbf{x},\mathbf{d}}\mathbf{P})}/_{\mathbf{x}}\right\}\left\{{}^{\mathbf{D}}/_{\mathbf{d}}\right\}$ (cf. Figure 4). In other words, the attest is available as the variable $\mathbf{x}$ in the program. There is no other way to obtain $attest$s, so only $\mathbf{P}$ can obtain an attest for $\mathbf{P}$. Note that we could not have modelled attests by just extending the signature because they are secret values that are provided only to specific syntactic components and the π-calculus does not support such constructs. The third argument, $\mathbf{D}$, represents data that $\mathbf{P}$ can use but is ignored for computing the attest.

The notation $\#_{\mathbf{x},\mathbf{d}}\mathbf{P}$ is shorthand for $\#new\,(\mathbf{x},\mathbf{d})\,\mathbf{P}$. We use new to bind variables $\mathbf{x}$ and $\mathbf{d}$ in $\mathbf{P}$ when computing the hash, such that equality of hashes can be taken up to renaming of bound variables, making variable names irrelevant.

The other two constructs model an authenticated form of communication. Given an attest and an expected identity, these operations can send or receive information provided that the attest of one party matches the identity the other party expects. We call these operations $\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M}, \mathbf{a}, \mathbf{h}\rangle$ and $\mathbf{N}^{\mathbf{auth}}(\mathbf{y}, \mathbf{h}, \mathbf{a})$ where $\mathbf{N}$ is the channel to send the message on, $\mathbf{M}$ is the term that gets sent, $\mathbf{y}$ is the variable that gets replaced by the received term, $\mathbf{a}$ is the attest or $anon$ if no attest is sent, and $\mathbf{h}$ is the expected identity of the other program or any if no attestation from the other party is required. The evaluation rules for this authenticated communication and $getAttest(\mathbf{x}, \mathbf{d}, \mathbf{D}).\mathbf{P}$ are shown in Figure 4.

The communication primitives are rather high-level: they are assumed to take care of defending against man-in-the-middle attacks and replay attacks so that protocols do not need to deal with this themselves. We could have alternatively used more low-level communication primitives from which we can build $\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M}, \mathbf{a}, \mathbf{h}\rangle$ and $\mathbf{N}^{\mathbf{auth}}(\mathbf{y}, \mathbf{h}, \mathbf{a})$, but that would only require us to prove the security of standard defences like nonces. We believe that our primitives allow the modeller to not have to deal with details that are irrelevant for the study of their protocols and that they align with the level of abstraction that the applied π-calculus normally operates at. Our primitives authenticate separately for every message sent. This inefficiency can be avoided by using an RA-authenticated channel to establish a shared key for communicating securely over unauthenticated channels.

B. Simple examples

In this section, we present two simple examples of programs making use of RA in π_{RA}. The first example demonstrates Secure Remote Execution (SRE) of a simple function and the second example demonstrates how RA could be used to make (non-interactive) zero-knowledge proofs.

SRE can be used for applications like secure distributed compilation [18], where computations are offloaded to other computers without trusting their operating system. For this example, we take a channel $\mathbf{N}$ and consider an attested component $getAttest(\mathbf{x}, \mathbf{d}, \mathbf{unit}).\mathbf{P}$ which performs a trivial computation and always returns $\mathbf{42}$:

$$\mathbf{P} = \overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{42}, \mathbf{x}, any\rangle.\mathbf{0}.$$

A program $\mathbf{Q}$ that uses the services of $\mathbf{P}$, can use RA to ensure that the computation is executed correctly:

$$\mathbf{Q} = \mathbf{N}^{\mathbf{auth}}(\mathbf{y}, \#_{\mathbf{x},\mathbf{d}}\mathbf{P}, anon).print(\mathbf{y}).\mathbf{0}$$

where $\mathbf{Q}$ uses the hash $\#_{\mathbf{x},\mathbf{d}}\mathbf{P}$ to make sure it communicates with $getAttest(\mathbf{x}, \mathbf{d}, \mathbf{unit}).\mathbf{P}$. Together, $\mathbf{P}$ and $\mathbf{Q}$ execute as follows:

$$\begin{aligned} & \mathbf{Q} \mid getAttest(\mathbf{x}, \mathbf{d}, \mathbf{unit}).\mathbf{P} \\ & \rightarrow \mathbf{Q} \mid \mathbf{P}\left\{{}^{attest(\#_{\mathbf{x},\mathbf{d}}\mathbf{P})}/_{\mathbf{x}}\right\}\left\{{}^{\mathbf{unit}}/_{\mathbf{d}}\right\} \\ & = \mathbf{N}^{\mathbf{auth}}(\mathbf{y}, \#_{\mathbf{x},\mathbf{d}}\mathbf{P}, anon).print(\mathbf{y}).\mathbf{0} \mid \\ & \quad \overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{42}, attest(\#_{\mathbf{x},\mathbf{d}}\mathbf{P}), any\rangle.\mathbf{0} \\ & \rightarrow print(\mathbf{42}).\mathbf{0} \mid \mathbf{0} \xrightarrow{\mathbf{p(42)}} \mathbf{0}. \end{aligned}$$

RA ensures that $\mathbf{Q}$ will only ever print $\mathbf{42}$, no matter what other program it is linked to, effectively achieving SRE.

Another example shows that remote attestation can support (non-interactive) zero-knowledge proofs for arbitrary criteria (cf. [19, Chapter 21]). Suppose program $\mathbf{P}$ wants to prove to $\mathbf{V}$ that it knows a secret value $\mathbf{s}$ that satisfies a criterion without leaking (any information about) $\mathbf{s}$. The idea is that $\mathbf{P}$ sends $\mathbf{s}$ to a third party, $\mathbf{R} = getAttest(\mathbf{x}, \mathbf{d}, \mathbf{unit}).\mathbf{R}'$: an attested component that validates $\mathbf{s}$ and sends out $true$ if $\mathbf{s}$ satisfies the criterion or $false$ otherwise. Because of RA, $\mathbf{V}$ can trust $\mathbf{R}$ to know whether $\mathbf{P}$ has the secret, while $\mathbf{P}$ can trust $\mathbf{R}$ to

$$\textsc{Attest}: \mathit{getAttest}(\mathbf{x},\mathbf{d},\mathbf{D}).\mathbf{P} \rightarrow \mathbf{P}\left\{^{\mathit{attest}(\#_{\mathbf{x},\mathbf{d}}\mathbf{P})}/_{\mathbf{x}}\right\}\left\{^{\mathbf{D}}/_{\mathbf{d}}\right\} \text{ if } \mathbf{D} \text{ is a ground term}$$

$$\textsc{AuthComm}: \frac{\forall \mathbf{R} \in \{\mathbf{P},\mathbf{Q}\} : (\mathbf{h_R} = \mathit{any} \text{ or } \mathit{attest}(\mathbf{h_R}) = \mathbf{a_R})}{\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M},\, \mathbf{a_P},\, \mathbf{h_Q}\rangle.\mathbf{P} \mid \mathbf{N}^{\mathbf{auth}}(\mathbf{y},\, \mathbf{h_P},\, \mathbf{a_Q}).\mathbf{Q} \rightarrow \mathbf{P} \mid \mathbf{Q}\left\{^{\mathbf{M}}/_{\mathbf{y}}\right\}}$$

Fig. 4: Extra evaluation rules of $\pi_{\mathbf{RA}}$.

forget the secret after validation. It does not even matter who controls $\mathbf{R}$ (either $\mathbf{P}$, $\mathbf{Q}$ or an untrusted third party), because the attested component's behaviour is fixed and independent of who created or hosts the attested component.

We take distinct channels $\mathbf{N_1}$ and $\mathbf{N_2}$ and define $\mathbf{R}'$ as:

$$\mathbf{R}' = \mathbf{N_1}^{\mathbf{auth}}(\mathbf{y},\, \mathit{any},\, \mathbf{x}).\overline{\mathbf{N_2}}^{\mathbf{auth}}\langle \mathit{checkCrit}(y),\, \mathbf{x},\, \mathit{any}\rangle.\mathbf{0}$$

where $\mathit{checkCrit}(\mathbf{s})$ is a function in the signature that is equal to true if $\mathbf{s}$ satisfies the criterion and false otherwise. The prover $\mathbf{P}$ and verifier $\mathbf{V}$ are defined as:

$$\begin{aligned}\mathbf{P} &= \overline{\mathbf{N_1}}^{\mathbf{auth}}\langle \mathbf{s},\, \mathit{anon},\, \#_{\mathbf{x},\mathbf{d}}\mathbf{R}'\rangle.\mathbf{0}\\ \mathbf{V} &= \mathbf{N_2}^{\mathbf{auth}}(\mathbf{y},\, \#_{\mathbf{x},\mathbf{d}}\mathbf{R}',\, \mathit{anon}).\\ &\qquad \mathit{if}\ \mathbf{y} = \mathit{true}\ \mathit{then}\ \textsc{Succ}\ \mathit{else}\ \textsc{Fail}.\end{aligned}$$

Together the components evaluate as follows:

$$\begin{aligned}&\mathbf{V} \mid \mathbf{P} \mid \mathit{getAttest}(\mathbf{x},\mathbf{d},\mathbf{unit}).\mathbf{R}'\\ &\rightarrow \mathbf{V} \mid \mathbf{P} \mid\\ &\quad (\mathbf{N_1}^{\mathbf{auth}}(\mathbf{y},\, \mathit{any},\, \mathit{attest}(\#_{\mathbf{x},\mathbf{d}}\mathbf{R}')).\\ &\qquad \overline{\mathbf{N_2}}^{\mathbf{auth}}\langle \mathit{checkCrit}(y),\, \mathit{attest}(\#_{\mathbf{x},\mathbf{d}}\mathbf{R}'),\, \mathit{any}\rangle.\mathbf{0})\\ &\rightarrow \mathbf{V} \mid \mathbf{0} \mid \overline{\mathbf{N_2}}^{\mathbf{auth}}\langle \mathit{checkCrit}(s),\, \mathit{attest}(\#_{\mathbf{x},\mathbf{d}}\mathbf{R}'),\, \mathit{any}\rangle.\mathbf{0}\\ &\rightarrow \mathit{if}\ \mathit{checkCrit}(\mathbf{s}) = \mathit{true}\ \mathit{then}\ \textsc{Succ}\ \mathit{else}\ \textsc{Fail} \mid \mathbf{0} \mid \mathbf{0}.\end{aligned}$$

The last line of this evaluation will further evaluate to $\textsc{Succ}$ if $\mathbf{s}$ satisfies the criterion and to $\textsc{Fail}$ otherwise. Note that $\mathit{checkCrit}(\mathbf{s})$ is a function in the signature and the equational theory defines it to be equal to true if $\mathbf{s}$ satisfies the criterion and to false otherwise. So, when $\mathit{checkCrit}(\mathbf{s})$ is sent, it is actually either true or false that is sent and no other information about $\mathbf{s}$ is included. This shows that, after a successful evaluation, $\mathbf{V}$ can trust that $\mathbf{P}$ has a suitable value, but has learned nothing else about the value. Also, $\mathbf{R}$ erased its copy of the value when it ends in an inactive program.

This shows that RA can be used to implement other cryptographic primitives like zero-knowledge proofs. We expect that RA could similarly be used to implement, e.g., secure multi-party computation.

C. Relation to concrete platforms

The primitives, $\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M},\, \mathbf{a},\, \mathbf{h}\rangle$, $\mathbf{N}^{\mathbf{auth}}(\mathbf{y},\, \mathbf{h},\, \mathbf{a})$ and $\mathit{getAttest}(\mathbf{x},\mathbf{d},\mathbf{D})$ from $\pi_{\mathbf{RA}}$, do not directly correspond to the primitives offered by systems with support for remote attestation. In this section, we argue that our model is realistic by explaining how primitives like $\mathit{getAttest}(\mathbf{x},\mathbf{d},\mathbf{D})$, $\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M},\, \mathbf{a},\, \mathbf{h}\rangle$ and $\mathbf{N}^{\mathbf{auth}}(\mathbf{y},\, \mathbf{h},\, \mathbf{a})$ correspond to primitives that are present in real systems.

The $\mathit{getAttest}(\mathbf{x},\mathbf{d},\mathbf{D})$ construct generates an attestation token that they can use to prove their identity in authenticated sends and receives. In our calculus based on rewrite rules, this allows us to remember what the attested component looks like at initialisation time. Many platforms never generate such a token, but as long as the components do not leak their attests, the attestation token can be regarded as a conceptual value that never leaves the component and hence doesn't require a runtime representation.

The $\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M},\, \mathbf{a},\, \mathbf{h}\rangle$ and $\mathbf{N}^{\mathbf{auth}}(\mathbf{y},\, \mathbf{h},\, \mathbf{a})$ constructs are quite abstract: as they combine attestation with communication and automatically protect against man-in-the-middle and replay attacks. Many systems do not offer secure communication primitives but provide more basic functionality, often a pair of primitives that allow (i) packaging a value with an attestation proof of its sender, and (ii) checking the authenticity of the package without direct access to the attest. Primitives behaving this way are mostly found in real systems at the moment of creating an attest and in these cases, unpackaged attests are never created. Some examples of functions on real platforms that have the same behaviour as a packaging function included in the attest request are "EREPORT" in Intel SGX [20], the "attest(d)" command with operand "d" in TAP [21], "ATTESTATION" in AMD SEV [22], "attest" in Keystone, "Attest" in the Komodo paper [23], "TPM2_Quote()" in the TPM [5], and the VRASED paper [24, p. 1431] describes a mechanism that has the properties of packaging. Sancus [3] uses a model based on symmetric keys and the attest is a key derived from the measurement of the attested component to communicate with the software provider. Since the key is not leaked during communication, the symmetric key also works as a packaged attest if the software provider is trusted, which is a necessary assumption to get RA between attested components in Sancus.

Instead of our authenticated sends and receives, we could have added such primitives to $\pi_{\mathbf{RA}}$ instead. The packaging and checking can be expressed in the equational theory of the applied π-calculus as the functions $\mathit{package}(\mathbf{M},\mathbf{b})$ and $\mathit{check}(\mathbf{pa},\mathbf{h},\mathbf{b})$, such that

$$\mathit{check}(\mathit{package}(\mathit{attest}(\mathbf{h}),\mathbf{b}),\mathbf{h},\mathbf{b}) = \mathit{true},$$

while everything else yields false. However, such primitives are more low-level and they would require typical RA-based protocols to take additional measures to prevent replay and man-in-the-middle attacks. Because we consider such prob-

lems to be standard and orthogonal to the use of remote attestation, we chose to use our more high-level $\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M}, \mathbf{a}, \mathbf{h}\rangle$ and $\mathbf{N}^{\mathbf{auth}}(\mathbf{y}, \mathbf{h}, \mathbf{a})$ in $\pi_{\mathbf{RA}}$, allowing protocols to focus on the use of RA and assume these more standard and well-understood problems to be dealt with in the background.

To implement the extra features that the Decent paper [12] uses, such as delegating authentication, dynamic verification and revocation, we also need to be able to create a token that shows that some enclave endorses something without being able to retrieve the attest from this. To support these features, we just need to add $package(\mathbf{M}, \mathbf{b})$ and $check(\mathbf{pa}, \mathbf{h}, \mathbf{b})$ to the signature. Also, keeping both authenticated receives and sends and packaging and checking at the same time gives the advantages of both simplicity and expressivity.

Lastly, $\pi_{\mathbf{RA}}$ allows some things that are not possible on some platforms supporting RA, such as nested attested components like

$$getAttest(\mathbf{x}, \mathbf{d}, \mathbf{D}).getAttest(\mathbf{y}, \mathbf{d}', \mathbf{D}').\mathbf{P}.$$

Furthermore, $\pi_{\mathbf{RA}}$ allows attestation tokens to be passed around, transmitted, and printed. A lot of systems do not allow nested attestation or transmission of attestation tokens, but protocols can support such systems by avoiding the use of these features. For example, our model of MAGE that will be demonstrated in the next section does not send attests across attested component boundaries. In fact, our proof shows that MAGE is even secure against attackers who can transmit attestation tokens.

IV. A demonstration: Formal analysis of MAGE

To demonstrate how $\pi_{\mathbf{RA}}$ can be used to reason about protocols using RA primitives, we formally define in our model the action of adding MAGE's mutual attestation features to any protocol. Specifically, we define MAGE as a compiler from π_{Actor} (cf. Section IV-A) to $\pi_{\mathbf{RA}}$ (cf. Section III-A) that uses the ideas of MAGE to resolve the circularity problem that occurs in mutual attestation. By proving that this compilation is correct and secure (in the sense of adapted versions of FAC and RrHC) and remarking that π_{Actor} can communicate securely along authenticated channels without circularity issues, we prove that the MAGE protocol succeeds at securely resolving the mutual authentication issues. Note here that π_{Actor} and its support for circular authenticated communication, are just as much a part of the security definition as the chosen secure compilation criteria. Our actor-based approach enables an approach similar to that of frameworks like EActors [25] that allow programmers to construct secure RA-based programs by programming in the comfort of a secure actor language without the manual use of RA.

A. The source language

The source language is an actor language inspired by the EActors framework [25] and the authentic execution framework of Noorman, Mühlberg and Piessens [26] where each actor contains a program and has a name. In this language, it is assumed that all programs know which name belongs to which program. It is also assumed that if two actors have the same name, then they must contain the same program. The source language π_{Actor} extends the applied π-calculus with actors:

$$\begin{aligned}
\mathsf{P} ::= \cdots \mid\ & \mathsf{actor}(\mathsf{n},\ \mathsf{P}) && \text{actor}\\
& \quad \text{(with } \mathsf{P} \text{ closed)} &&\\
& \mathsf{runAct}(\mathsf{n},\ \mathsf{P},\ \mathsf{P}_{start}) && \text{initialised/running actor}\\
& \overline{\mathsf{N}}_{id_1 \to id_2}\langle \mathsf{M}\rangle.\mathsf{P} && \text{output with identification}\\
& \mathsf{N}_{id_2 \to id_1}(\mathsf{x}).\mathsf{P} && \text{input with identification}\\
& \quad \text{with } id_1 := self \mid anon \text{ and } id_2 := \mathsf{n} \mid any &&
\end{aligned}$$

The construct $\mathsf{actor}(\mathsf{n},\ \mathsf{P})$ represents an actor that has not yet started execution and $\mathsf{runAct}(\mathsf{n},\ \mathsf{P},\ \mathsf{P}_{start})$ denotes an actor that is already initialised with name n and starting program P_{start} and currently contains the program P. Programs in π_{Actor} do not contain running actors in the beginning; these are obtained by evaluating $\mathsf{actor}(\mathsf{n},\ \mathsf{P})$. Additionally, we have the safe output and input, $\overline{\mathsf{N}}_{id_1 \to id_2}\langle \mathsf{M}\rangle.\mathsf{P}$ and $\mathsf{N}_{id_2 \to id_1}(\mathsf{x}).\mathsf{P}$, where the id_1 is the identity the program uses and id_2 is the expected identity of the other party. The program's identity id_1 can be $self$ (referring to the name of the running actor) or $anon$ (anonymous). Safe inputs and outputs outside actors can not use $id_1 = self$, hence they are always anonymous. The identity id_2 is either the expected name of the other party or any, allowing any party.

There are also some extra congruence rules for π_{Actor}:

$$\mathsf{runAct}(\mathsf{n},\ \mathsf{P} \mid \mathsf{Q},\ \mathsf{P_s}) \equiv \mathsf{runAct}(\mathsf{n},\ \mathsf{P},\ \mathsf{P_s}) \mid \mathsf{runAct}(\mathsf{n},\ \mathsf{Q},\ \mathsf{P_s})$$

$$\frac{\mathsf{P} \equiv \mathsf{P}'}{\mathsf{runAct}(\mathsf{n},\ \mathsf{P},\ \mathsf{P}_{start}) \equiv \mathsf{runAct}(\mathsf{n},\ \mathsf{P}',\ \mathsf{P}_{start})}.$$

The first rule makes sure that communication within a single actor can go through. The second rule states that actors with the same name and starting program containing congruent programs are congruent, as expected. A similar rule is not present for the starting program $\mathsf{P_s}$ in $\mathsf{runAct}(\mathsf{n},\ \mathsf{P},\ \mathsf{P_s})$ or $\mathsf{actor}(\mathsf{n},\ \mathsf{P_s})$ because RA will attest a specific syntactic identity of the program.

The evaluation rules of π_{Actor} can be found in Figure 6 and they depend on the labelled transition rules in Figure 5. The rule InitActor initialises an actor (emitting an empty label) and EvalActor makes transition rules apply within actors as well. The most interesting rule is SafeComm which deals with communication. This rule uses $\overset{\mathsf{I}}{\hookrightarrow}$ (cf. Figure 5) to handle things like replacing $self$ by the correct name (done by $labelOut(\mathsf{a}, \mathsf{n})$ in ActorOut) and getting input or outputs requests through some layers of actors. The rules reflect our earlier explanation about the identities in $\overline{\mathsf{N}}_{id_1 \to id_2}\langle \mathsf{M}\rangle.\mathsf{P}$ and $\mathsf{N}_{id_2 \to id_1}(\mathsf{x}).\mathsf{P}$.

To demonstrate π_{Actor}, we reconsider the Secure Remote Execution example from Section III-B, this time in π_{Actor}. Take a channel N and define $\mathsf{P} = \overline{\mathsf{N}}_{self \to any}\langle 42\rangle.\mathbf{0}$ (a program that computes 42) and $\mathsf{Q} = \mathsf{N}_{\mathsf{n} \to anon}(\mathsf{y}).print(\mathsf{y})$ (a program that

$$\textsc{SafeOutput} : \overline{\mathsf{N}}_{id_1 \to id_2}\langle \mathsf{M}\rangle.\mathsf{P} \xrightarrow{\overline{\mathsf{N}}_{id_1 \to id_2}\langle \mathsf{M}\rangle} \mathsf{P}$$

$$\textsc{SafeInput} : \mathsf{N}_{id_2 \to id_1}(\mathsf{y}).\mathsf{P} \xrightarrow{\mathsf{N}_{id_2 \to id_1}(\mathsf{M})} \mathsf{P}\left\{{}^{\mathsf{M}}/_{\mathsf{y}}\right\}$$

$$\textsc{ActorOut} : \frac{\mathsf{P} \overset{\mathsf{a}}{\hookrightarrow} \mathsf{P}'}{\mathsf{runAct}(\mathsf{n},\ \mathsf{P},\ \mathsf{P}_{start}) \xrightarrow{labelOut(\mathsf{a},\mathsf{n})} \mathsf{runAct}(\mathsf{n},\ \mathsf{P}',\ \mathsf{P}_{start})}$$

with

$$\begin{aligned} selfToName(self, \mathsf{n}) &= \mathsf{n} \\ selfToName(id, \mathsf{n}) &= id \\ labelOut(\overline{\mathsf{N}}_{id_1 \to id_2}\langle \mathsf{M}\rangle, \mathsf{n}) &= \overline{\mathsf{N}}_{selfToName(id_1,\mathsf{n}) \to id_2}\langle \mathsf{M}\rangle \\ labelOut(\mathsf{N}_{id_2 \to id_1}(\mathsf{M}), \mathsf{n}) &= \mathsf{N}_{id_2 \to selfToName(id_1,\mathsf{n})}(\mathsf{M}) \end{aligned}$$

Fig. 5: The labelled transitions denoted by $\overset{\mathsf{l}}{\hookrightarrow}$ and the function $selfToName(id, n)$ and $labelOut(label, n)$.

$$\textsc{InitActor} : \mathsf{actor}(\mathsf{n},\ \mathsf{P}) \to \mathsf{runAct}(\mathsf{n},\ \mathsf{P},\ \mathsf{P})$$

$$\textsc{EvalActor} : \frac{\mathsf{P} \overset{\mathsf{a}}{\to} \mathsf{P}'}{\mathsf{runAct}(\mathsf{n},\ \mathsf{P},\ \mathsf{P}_{start}) \overset{\mathsf{a}}{\to} \mathsf{runAct}(\mathsf{n},\ \mathsf{P}',\ \mathsf{P}_{start})}$$

$$\textsc{SafeComm} : \frac{\begin{array}{c}\mathsf{P} \xrightarrow{\overline{\mathsf{N}}_{id_{\mathsf{P}1} \to id_{\mathsf{P}2}}\langle \mathsf{M}\rangle} \mathsf{P}' \qquad \mathsf{Q} \xrightarrow{\mathsf{N}_{id_{\mathsf{Q}2} \to id_{\mathsf{Q}1}}(\mathsf{M})} \mathsf{Q}' \\ (id_{\mathsf{P}2} = any \text{ or } id_{\mathsf{P}2} = id_{\mathsf{Q}1}) \\ (id_{\mathsf{Q}2} = any \text{ or } id_{\mathsf{Q}2} = id_{\mathsf{P}1})\end{array}}{\mathsf{P} \mid \mathsf{Q} \to \mathsf{P}' \mid \mathsf{Q}'}.$$

Fig. 6: The extra rules of the transition relation of π_{Actor}.

wishes to makes use of these services). The program Q can safely make use of a protected version of P in an actor n: $\mathsf{Q} \mid \mathsf{actor}(\mathsf{n},\ \mathsf{P})$. It will evaluate as follows:

$$\begin{aligned} &\mathsf{N}_{\mathsf{n} \to anon}(\mathsf{y}).print(\mathsf{y}) \mid \mathsf{actor}(\mathsf{n},\ \mathsf{P}) \to \\ &\mathsf{N}_{\mathsf{n} \to anon}(\mathsf{y}).print(\mathsf{y}) \mid \mathsf{runAct}(\mathsf{n},\ \overline{\mathsf{N}}_{self \to any}\langle 42\rangle.\mathbf{0},\ \mathsf{P}) \to \\ &print(42) \mid \mathsf{runAct}(\mathsf{n},\ \mathbf{0},\ \mathsf{P}) \xrightarrow{\mathsf{p}(42)} \mathbf{0} \mid \mathsf{runAct}(\mathsf{n},\ \mathbf{0},\ \mathsf{P}). \end{aligned}$$

Again, notice that the program $\mathsf{N}_{\mathsf{n} \to anon}(\mathsf{y}).print(\mathsf{y})$ will only ever print 42, even if the channel N is public. This is because contexts cannot define actors with the same name n but a different program, so whichever actor Q communicates with will have P as its starting program. Note that Q is assumed to know the link between n and P, so $\mathsf{actor}(\mathsf{n},\ \mathsf{P})$ is part of the trusted code base but may be running on an untrusted system.

B. The compiler

We can now formalise MAGE as a compiler from π_{Actor} to $\pi_{\mathbf{RA}}$. Recall from Section II-c that MAGE relies on some properties of the specific implementation of the hash used in Intel SGX, namely that the hash of the concatenation

$$\begin{aligned} [\![print(\mathsf{M}).\mathsf{P}]\!]_{\mathbf{idT},\mathbf{x}} &= print(\mathbf{M}).[\![\mathsf{P}]\!]_{\mathbf{idT},\mathbf{x}} \\ [\![\mathsf{actor}(\mathsf{n},\ \mathsf{P})]\!]_{\mathbf{idT},\mathbf{x}} &= getAttest(\mathbf{y}, \mathbf{d}, \mathbf{unit}).[\![\mathsf{P}]\!]_{\mathbf{idT},\mathbf{y}} \\ &\quad \text{with } \mathbf{y} \text{ and } \mathbf{d} \text{ fresh} \\ [\![\overline{\mathsf{N}}_{id_1 \to id_2}\langle \mathsf{M}\rangle.\mathsf{P}]\!]_{\mathbf{idT},\mathbf{x}} &= \\ \overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M},\ slfToA(id_1, x),\ ext(\mathbf{idT}(id_2), \mathbf{idT})\rangle.&[\![\mathsf{P}]\!]_{\mathbf{idT},\mathbf{x}} \\ [\![\mathsf{N}_{id_2 \to id_1}(\mathsf{y}).\mathsf{P}]\!]_{\mathbf{idT},\mathbf{x}} &= \\ \mathbf{N}^{\mathbf{auth}}(\mathbf{y},\ ext(\mathbf{idT}(id_2), \mathbf{idT}),\ slfToA(id_1, x)).&[\![\mathsf{P}]\!]_{\mathbf{idT},\mathbf{x}} \end{aligned}$$

$$\text{with } slfToA(id_1, \mathbf{a}) = \begin{cases} \mathbf{a} & \text{if } id_1 = self \\ anon & \text{otherwise} \end{cases}.$$

Fig. 7: The compiler, parametrized by $\mathbf{idT}$ and k.

of A and the preprocessed table of identities, idT, can be calculated from the hash of A and the content of idT. To model this property, we add a function $ext((\mathbf{h}, \mathbf{D}))$ to the signature of $\pi_{\mathbf{RA}}$ with $ext(\#new\,\mathbf{z}\,\mathbf{P}, \mathbf{idT}) = \#\mathbf{P}\left\{{}^{\mathbf{idT}}/_{\mathbf{z}}\right\}$ and $ext(any, \mathbf{D}) = any$ as extra equational theory. This function can take a hash of a program and a term and return the hash of the program with the term substituted. This is similar to the hash of concatenated programs mentioned in Section II-c.

Recall from Section II-c that MAGE relies on the table of identities that was concatenated at the end. However, before showing how we calculate these preprocessed identities, we define the first phase of the compilation which is parametric on this identities table (cf. Figure 7). This is parametric which means that we can use identities of this kind of compilation for the table of identities and by replacing the placeholder $\mathbf{idT}$ by the preprocessed table we get a compiled program that can use $\mathbf{idT}$.

Most of the rules in this compiler are fairly straightforward and not mentioned in Figure 7. The first rule is an example of what the omitted rules look like. There is no rule for running actors because programs in π_{Actor} do not contain running actors initially. Uninitialised actors are compiled to $getAttest(\mathbf{x}, \mathbf{d}, \mathbf{D})$-regions. Output and input are compiled to authenticated sends and receives with the right identities table and the attest is either $\mathbf{x}$ for $self$ (to be replaced later by the actual attest) or $anon$ for $anon$.

Next, this first phase parametric in $\mathbf{idT}$ is used to create the identities table with the function $cH(\mathsf{P})$ (cf. Figure 8) where using a variable in the place of $\mathbf{idT}$ disregards the hashes of other programs in the code (just remembers the names). The function $cH(\mathsf{P})$ then collects the hashes of these compilations without the identities table in a dictionary mapping names of actors to hashes of their code.

Using these concepts the actual compilation can then be defined as using the compilation dependent on the identities table with the preprocessed identities table as an argument or

$$cH(\mathsf{P}) = \begin{cases} \emptyset & \text{if } \mathsf{P} = \mathbf{0} \\ \{(\mathsf{n}, \#_{z,x,d}[\![\mathsf{P}_{start}]\!]_{d,x})\} \cup cH(\mathsf{P}_{start}) & \text{if } \mathsf{P} = \mathsf{actor}(\mathsf{n},\ \mathsf{P}_{start}) \text{ or } \mathsf{P} = \mathsf{runAct}(\mathsf{n},\ \mathsf{P}',\ \mathsf{P}_{start}) \\ cH(\mathsf{Q}) & \text{if } \mathsf{P} = \overline{\mathsf{N}}_{id_1 \to id_2}\langle \mathsf{M} \rangle.\mathsf{Q} \text{ or } \mathsf{P} = \mathsf{N}_{id_2 \to id_1}(\mathsf{x}).\mathsf{Q} \text{ or } \ldots \\ cH(\mathsf{Q}) \cup cH(\mathsf{R}) & \text{if } \mathsf{P} = \mathsf{Q} \mid \mathsf{R} \text{ or } \mathsf{P} = \mathsf{Q} + \mathsf{R} \text{ or } \mathsf{P} = \mathit{if}\ \mathsf{M} = \mathsf{N}\ \mathit{then}\ \mathsf{Q}\ \mathit{else}\ \mathsf{R} \end{cases}.$$

Fig. 8: The function $cH(\mathsf{P})$ compiles a table of the hashes of compiled actors in P.

in symbols:

$$[\![\mathsf{P}]\!] = [\![\mathsf{P}]\!]_{cH(\mathsf{P}),\blacksquare}$$

where $\blacksquare$ denotes a fixed choice of variable, but $\blacksquare$ is never used as a variable name by other programs.

Despite the high-level nature of our source and target languages, our model still faithfully captures MAGE's vital parts: the preprocessed table of identities which is generated at compile time and the reconstruction of counterparty identities at runtime by combining the preprocessed identities with the full table. Furthermore, communication in the source which is done with authenticated communication primitives without circularity problems is compiled to communication primitives that use attests to obtain authentication and MAGE's solution to solve the circularity.

As explained before, our compiler does not model the MAGE protocol as a standalone (set of) communication steps but as a procedure that translates an existing program from an abstract idealised setting into a more concrete setting. The compiler does this by implementing idealized communication primitives (that natively allow circular attestation) in terms of more elementary primitives and using the MAGE protocol to resolve the circularity. This is a good way to model the MAGE protocol because (1) the protocol is parametric in a program-specific value (the identities table), so it is natural to consider it as a translation/compiler, (2) the two languages are very close but differ precisely in how they deal with circular attestation, the central issue that MAGE addresses, (3) the compiler does not perform any interesting form of translation, except for applying MAGE to implement circular attestation in terms of the target language's more elementary primitives and (4) this model allows us to analyse the MAGE protocol's security by connecting security properties of the original and translated version of an abstract program, i.e. as a secure compilation result. This approach is similar to existing approaches in the field of cryptography that connect real/ideal implementations of a protocol (like the universal composability approach [14]) and has been used before to model and analyse cryptographic protocols [27].

C. Attacker models

As mentioned in the introduction of Section IV, the secure compilation results for this compiler constitute a security property for the MAGE protocol. However, the chosen attacker models of the source and target are also an integral part of what these results mean, so we define attackers in the source and the target language next. In the target, it is assumed that initially neither context nor program contains any attests which makes sure that attests can only be obtained by evaluating $getAttest(\mathbf{x}, \mathbf{d}, \mathbf{D}).\mathbf{P}$ which is realistic because RA systems only hand out attests for a program's own code.

In the source, there are initially no running actors present and it is assumed that the context cannot add more distinct $actor$s. The context can only add extra copies of $actor$s that are already present in the program. For example, the context of $\mathsf{actor}(\mathsf{n},\ \mathsf{P})$ cannot contain $\mathsf{actor}(\mathsf{m},\ \mathsf{Q})$ or $\mathsf{actor}(\mathsf{n},\ \mathsf{Q})$ with $\mathsf{P} \neq \mathsf{Q}$, but it can contain another $\mathsf{actor}(\mathsf{n},\ \mathsf{P})$.

The fact that source contexts can contain copies of actors in the program being compiled formally reflects an important caveat in the attacker model of MAGE. This caveat was not mentioned in the original paper and was uncovered by us during our formal analysis. In section IV-E, we will explain this in more detail.

Currently, we do not allow $actor$s with different names but the same program because our current compiler would not be able to distinguish them. This could be resolved easily by making the compiler include the name of the actor in the compiled actor's code.

D. Security results

In this section, we present the security theorems that were shown. We chose FAC because it is a well-established criterion, however, FAC still allows some bugs and sometimes obscures which security is actually provided [17]. For this reason, we also proved a variant of RrHC. RrHC is not as well established as FAC, but it is the strongest secure compilation criterion proposed to date, the resulting security is clearer and the preservation side of FAC directly follows from RrHC. On top of that, a strong criterion like our variant of RrHC allows us to conclude that our model, and thus the existing MAGE framework [9], provides strong security guarantees to its users. Intuitively, these secure compilation criteria show that MAGE works securely because the source language π_{Actor} has authenticated mutual communication built-in yet attackers against compiled programs do not have more power (in the sense of RrHC or FAC), hence the way authenticated mutual communication is implemented in the target is secure.

We now present our variant of RrHC:

Theorem 1. *Suppose τ_{k} is defined as in Figure 9 and τ_{k} maps traces pointwise, then*

$$\forall \mathbf{idT}, \mathsf{k}, \mathbf{C} : \exists \mathsf{C_S} : \forall \mathsf{P} : \mathbf{idT} = cH(\mathsf{P}) \wedge \mathsf{k} \notin \mathbf{C}, \mathsf{P}$$
$$\implies \tau_{\mathsf{k}}(behav(\mathbf{C}[\![\mathsf{P}]\!])) = behav(\mathsf{C_S}[\mathsf{P}]).$$

$$\tau_{\mathsf{k}}(\mathbf{M}) = \begin{cases} \#\mathbf{P} & \text{if } M = \#\mathbf{P} \\ \#(\mathsf{k}, \tau_{\mathsf{k}}(\mathbf{h})) & \text{if } \mathbf{M} = \mathit{attest}(\mathbf{h}) \\ \mathsf{f}(\tau_{\mathsf{k}}(\mathbf{M_1}), ..., \tau_{\mathsf{k}}(\mathbf{M_n})) & \text{if } \mathbf{M} = \mathbf{f}(\mathbf{M_1}, ..., \mathbf{M_n}) \\ \mathsf{M} & \text{otherwise} \end{cases}$$

where p and ui are treated as functions in the signature.

Fig. 9: Definition of τ_{k}

The map $[\![\cdot]\!]$ denotes the compiler defined in Section IV-B.

Here $\mathsf{k} \notin \mathbf{C}, \mathsf{P}$ means that the variable k is not present in $\mathbf{C}$ or P. Intuitively, the theorem states that given only the target context $\mathbf{C}$ and the identities of the actors in the linking program (or the identities table of the linking program), $cH(\mathsf{P})$, there exists a context in the source language, $\mathsf{C_S}$, such that $\mathbf{C}[[\![\mathsf{P}]\!]]$ and $\mathsf{C_S}[\mathsf{P}]$ have the same behaviour. Note that this is a weaker criterion than robust relational hyperproperty characterisation (RrHC) because in RrHC no knowledge of the linking program, P, is used to construct the source context whereas here $cH(\mathsf{P})$ can be used. Abate et al. [16] also present a weaker non-relational criterion similar to RrHC where the source context can depend on the linked program. This property is called RHC and it is implied by our property (cf. Theorem 1). We interpret our property (cf. Theorem 1) as a variant of the robust relational hyperproperty characterisation (RrHC) and it essentially means that the choice of $\mathsf{C_S}$ may depend on $cH(\mathsf{P})$, but not on other knowledge about P. The reason for this criterion will be explained below. The quantification over k works similarly but is not a real limitation: our back-translation simply requires a name k which does not appear in either $\mathbf{C}$ and P, but it is clear that such a k always exists.

The map τ_{k} is needed to solve the problem that the target language can print attests, meaning $\mathit{print}(\mathit{attest}(\mathbf{h}))$ is possible in a target program, but attests are not defined in the source language. This is a problem because RrHC depends on the equality of traces, but a source program can never have the same trace as a target program that prints an attest. To solve this, we state RrHC up to an injective mapping of traces, τ_{k}, that maps $\mathit{attest}(\mathbf{h})$ to $\#(\mathsf{k}, \mathbf{h})$ and acts as the identity otherwise. To ensure the injectivity of τ_{k}, the variable k is chosen such that it is not present in the source program or the target context that is defined in the definition of RrHC. Because τ_{k} is an injective map, this variant of RrHC still yields a useful result, although it is weaker than actual RrHC. Intuitively, it is clear that the context can print more things, but that the actual security is not weakened by this adaptation.

Our adaptation of RrHC also allows the choice of $\mathsf{C_S}$ to depend on $\mathbf{idT} = cH(\mathsf{P})$ (making it weaker than the original RrHC), such that communication with compiled actors in $[\![\mathsf{P}]\!]$ can be faithfully modelled in $\mathsf{C_S}$. This condition is needed because the set of valid attackers in the source depends on the actors present in the linked program, P (the only actors in the source contexts have to be copies of actors in P). It is important to make sure that $\mathsf{C_S}$ is a valid source context, hence the actors in P have to be known when choosing $\mathsf{C_S}$. Related to this problem is the problem in the target that the content of attested components at initialisation is known by everyone (through the identities table in the target) which means any attacker can distinguish whether an attested component is initialised with a certain hash. We show an example of this in Section 6.3 of the supplementary material. This means that asking RrHC for these actors is unreasonable because communication of a compiled actor with the context might behave differently based on the hash of the resulting attested component. The property is still very useful despite this limitation because this can be interpreted as a security result about programs (clients) communicating with specific actors and not a security theorem about the actors themselves. The goal of the compilation should be that these clients stay secure, meaning their secrets do not leak and they get back the expected values from the actors they interacted with.

The second theorem is the adapted FAC:

Theorem 2. *If P and Q are source programs with $cH(\mathsf{P}) = cH(\mathsf{Q})$, then*

$$\mathsf{P} \simeq_{\mathsf{ctx}} \mathsf{Q} \iff [\![\mathsf{P}]\!] \simeq_{\mathbf{ctx}} [\![\mathsf{Q}]\!]$$

holds, where $[\![\cdot]\!]$ is the compiler defined in Section IV-B.

This theorem is an adapted version of FAC because we assume $cH(\mathsf{P}) = cH(\mathsf{Q})$ for similar reasons as before. This means that P and Q contain the same actors initially. Note that contextual equivalence, $\mathsf{P} \simeq_{\mathsf{ctx}} \mathsf{Q}$, is only a useful concept if the actors in P and Q are the same because then the valid contexts are the same for both programs. The fact that the hashtables are known (similarly to the problem with RrHC) also makes distinguishing actors trivial (cf. Section 6.3 of the supplementary material). We could also state this theorem as regular FAC by using a different definition for contextual equivalence in the source, but this notation is clearer on the limitations.

E. Copy attack

In Section IV-C, we mention a caveat in the security of MAGE, namely that MAGE does not distinguish copies of enclaves. We call an attack that is possible by using such copies a copy attack. Below we demonstrate in our formalism why not allowing copies of actors in the source attacker (the formalisation of this attack) can break full abstraction by mounting such a copy attack. Take R to be an actor that outputs 5 once and 6 on every subsequent time:

$$\mathsf{R} = \mathsf{actor}(\mathsf{n}, \overline{\mathsf{N}}_{self \to any}\langle 5\rangle.!\,(\overline{\mathsf{N}}_{self \to any}\langle 6\rangle.\mathbf{0})).$$

Take Q to be a program that accepts two messages from the actor n and diverges if they are both 5:

$$\mathsf{Q} = \mathsf{N}_{\mathsf{n}\to anon}(\mathsf{y}').\mathsf{N}_{\mathsf{n}\to anon}(\mathsf{y}'').\text{if } \mathsf{y}' = \mathsf{y}'' = 5 \text{ then } \omega \text{ else } \mathbf{0}.$$

Here, ω denotes the diverging program. Take $\mathsf{P} = \mathsf{N}_{\mathsf{n}\to anon}(\mathsf{y}').\mathsf{N}_{\mathsf{n}\to anon}(\mathsf{y}'').\mathbf{0}$. The author of Q might believe that $\mathsf{Q} \mid \mathsf{R}$ and $\mathsf{P} \mid \mathsf{R}$ are contextually equivalent because all

communications can only happen with R and there is only one R, so the equality check in Q will always fail, resulting in Q converging every time. However, this is only true if the attacker model in the source does not allow copies of existing actors (so the attacker cannot contain any actor).

Now, we compile $\mathsf{R} \mid \mathsf{Q}$ and $\mathsf{R} \mid \mathsf{P}$ to get the following programs:

$$
\begin{aligned}
&[\![\mathsf{R} \mid \mathsf{Q}]\!] = \\
&\mathit{getAttest}(\mathbf{x}, \mathbf{d}, \mathbf{unit}). \\
&\quad \overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{5},\ \mathbf{x},\ \mathit{any}\rangle.!\,(\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{6},\ \mathbf{x},\ \mathit{any}\rangle) \mid \\
&\mathbf{N}^{\mathbf{auth}}(\mathbf{y}',\ \mathit{ext}(\mathbf{idT}(\mathsf{n}), \mathbf{idT}),\ \mathit{anon}). \\
&\quad \mathbf{N}^{\mathbf{auth}}(\mathbf{y}'',\ \mathit{ext}(\mathbf{idT}(\mathsf{n}), \mathbf{idT}),\ \mathit{anon}). \\
&\qquad \text{if } \mathbf{y}' = \mathbf{y}'' = \mathbf{5} \text{ then } \boldsymbol{\omega} \text{ else } \mathbf{0} \\
&\qquad \text{with } \mathbf{idT} = cH(\mathsf{R} \mid \mathsf{Q}) = cH(\mathsf{R})
\end{aligned}
$$

and

$$
\begin{aligned}
&[\![\mathsf{R} \mid \mathsf{P}]\!] = \\
&\mathit{getAttest}(\mathbf{x}, \mathbf{d}, \mathbf{unit}). \\
&\quad \overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{5},\ \mathbf{x},\ \mathit{any}\rangle.!\,(\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{6},\ \mathbf{x},\ \mathit{any}\rangle) \mid \\
&\mathbf{N}^{\mathbf{auth}}(\mathbf{y}',\ \mathit{ext}(\mathbf{idT}(\mathsf{n}), \mathbf{idT}),\ \mathit{anon}). \\
&\quad \mathbf{N}^{\mathbf{auth}}(\mathbf{y}'',\ \mathit{ext}(\mathbf{idT}(\mathsf{n}), \mathbf{idT}),\ \mathit{anon}).\mathbf{0} \\
&\qquad \text{with } \mathbf{idT} = cH(\mathsf{R} \mid \mathsf{P}) = cH(\mathsf{R}).
\end{aligned}
$$

Here, you see that the compilation starts by making a table of all the preprocessed identities of the present actors (this is $cH(\mathsf{R} \mid \mathsf{Q})$ in the first and $cH(\mathsf{R} \mid \mathsf{P})$ in the second case which are both equal to $cH(\mathsf{R})$). Subsequently, an actor is replaced by an attested region, $\mathit{getAttest}(\mathbf{x}, \mathbf{d}, \mathbf{D})$ and communication in the source is replaced by authenticated communication where $\mathbf{x}$ is used to be replaced by the attest if the source communication referred to self. The last important part of this compilation is the fact that if the source communication referred to n, then the authenticated communication demands the hash, $\mathit{ext}(\mathbf{idT}(\mathsf{n}), \mathbf{idT})$ with $\mathbf{idT}$ being the preprocessed table of identities. This corresponds to how in MAGE the actual hash of a program n which uses the common hashtable is constructed from the nth entry of $\mathbf{idT}$ and $\mathbf{idT}$ itself.

Now, we note that these compiled programs, $[\![\mathsf{R} \mid \mathsf{Q}]\!]$ and $[\![\mathsf{R} \mid \mathsf{P}]\!]$, can be distinguished by the following attacker:

$$
\begin{aligned}
&\mathit{getAttest}(\mathbf{x}, \mathbf{d}, \mathbf{unit}). \\
&\quad \overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{5},\ \mathbf{x},\ \mathit{any}\rangle.!\,(\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{6},\ \mathbf{x},\ \mathit{any}\rangle) \mid [\cdot]
\end{aligned}
$$

This attacker can distinguish them because in the first case $[\![\mathsf{Q}]\!]$ can diverge by communicating once with $[\![\mathsf{R}]\!]$ and once with the attacker to get $y' = y'' = 5$, while the $[\![\mathsf{P}]\!]$ can only converge. Distinguishing these compiled programs works because the attested region in the attacker produces exactly the same attest as the compilation of R, so there is no way to distinguish these attested regions. This means that the subsequent communications might each be with another instance of the compilation of R, so both queries might return 5 which makes the check in the compilation of Q succeed, hence it diverges. On the other hand, the second program can never diverge, so the attacker can distinguish them. This example shows that the compiler is not fully abstract if the attacker model in the source does not allow the attacker to use copies of actors from the linked program.

In this case, we can easily fix the problem by setting up a session (e.g., by Diffie-Hellman key exchange) before we start communicating. However, the solution is not as simple if more than two actors are involved. Take R to be the same as before and take Q and Q' to be actors, with names m and m', that accept a message from actor n, add 2 to the messages and send it on to our client:

$$
\begin{aligned}
\mathsf{Q} &= \mathsf{actor}(\mathsf{m},\ \mathsf{N}_{\mathsf{n}\to anon}(\mathsf{y}).\mathit{print}(1).\overline{\mathsf{N}}_{\mathit{self}\to any}\langle \mathsf{y}+2\rangle) \\
\mathsf{Q}' &= \mathsf{actor}(\mathsf{m}',\ \mathsf{N}_{\mathsf{n}\to anon}(\mathsf{y}).\mathit{print}(2).\overline{\mathsf{N}}_{\mathit{self}\to any}\langle \mathsf{y}+2\rangle)
\end{aligned}
$$

where the print makes sure these actors do not contain the same program. We look at the programs

$$\mathsf{P} = \mathsf{N}_{\mathsf{m}\to anon}(\mathsf{y}).\mathsf{N}_{\mathsf{m}'\to anon}(\mathsf{y}').\text{if } \mathsf{y} = \mathsf{y}' = 7 \text{ then } \omega \text{ else } \mathbf{0}$$

and $\mathsf{P}' = \mathsf{N}_{\mathsf{m}\to anon}(\mathsf{y}).\mathsf{N}_{\mathsf{m}'\to anon}(\mathsf{y}').\mathbf{0}$. If the attacker cannot use copies of actors, specifically copies of R, then $\mathsf{R} \mid \mathsf{Q} \mid \mathsf{Q}' \mid \mathsf{P}$ cannot be distinguished from $\mathsf{R} \mid \mathsf{Q} \mid \mathsf{Q}' \mid \mathsf{P}'$, because the equality check in P will always fail as all values need to come from R. However, if copies are allowed, then the attacker $\mathsf{R} \mid [\cdot]$ can distinguish the two programs because now Q and Q' can get their value from different instances of R, so both messages can be 5 and this makes the equality in P succeed, hence $\mathsf{R} \mid \mathsf{R} \mid \mathsf{Q} \mid \mathsf{Q}' \mid \mathsf{P}$ can diverge, while $\mathsf{R} \mid \mathsf{R} \mid \mathsf{Q} \mid \mathsf{Q}' \mid \mathsf{P}'$ cannot. This shows that the attacker can indeed distinguish the two programs.

In the case of one actor, the solution was quite simple, however, making sure that no copies are allowed in a system with several actors is more complicated than simply setting up a session as before. If several parties are using the same enclave, then they need to coordinate to make sure that they are using the same enclave and not different copies. There are some solutions to do this, for example, an authenticated group key exchange [28] or letting every program keep a table of public keys associated with private keys in each actor, but those are all non-trivial and MAGE does not seem intended to prevent such attacks.

V. Proof

This section discusses our proofs of Theorem 1 and 2 and some of the technical difficulties encountered. This section is not essential to understand the concepts and can be skipped. We refer the interested reader to the supplementary material for the full proof.

A. Equivalence reflection for FAC

The equivalence reflection of Theorem 2 ('$\Longleftarrow$') depends on the correctness of the compiler rather than the security. We show the correctness of the compiler by constructing a bisimulation $\mathfrak{R}$ such that $\mathsf{P}\,\mathfrak{R}\,[\![\mathsf{P}]\!]$ holds for all compiled programs. The appropriate bisimulation is presented in the supplementary material along with a proof that it is a bisimulation, but will

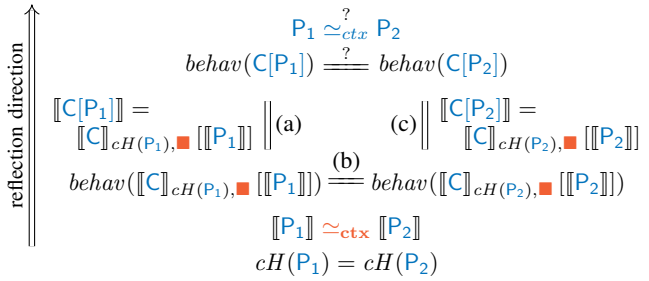

Fig. 10: Diagram of the proof of equivalence reflection of FAC. This diagram is adapted from the diagram in [29].

not be discussed here. The existence of this bisimulation shows that $behav(\mathsf{P}) = behav([\![\mathsf{P}]\!])$ holds for each source program P, because if P evaluates with a label, then $[\![\mathsf{P}]\!]$ can also evaluate with the same label (because they are related by a bisimulation) and the results of the evaluations are still related by the bisimulation. This reasoning holds for any trace, hence $[\![\mathsf{P}]\!]$ can evaluate with each of the traces that P can evaluate with and vice versa.

To derive equivalence reflection from this behavioural equality, the condition $cH(\mathsf{P}) = cH(\mathsf{Q})$ is needed because these identities tables limit the set of valid source contexts and they dictate how the context should be compiled (that is with which identities table as an argument). The scheme of this proof can be seen in Figure 10 where it is still important to note that $[\![\mathsf{C}[\mathsf{P}]]\!] = [\![\mathsf{C}]\!]_{cH(\mathsf{P}),\blacksquare}\,[[\mathsf{P}]]$ holds for each source program P because we know that $[\![\cdot]\!]_{\mathbf{idT},\mathbf{x}}$ compiles piece by piece, we know that $[\![\mathsf{P}]\!] = [\![\mathsf{P}]\!]_{cH(\mathsf{P}),\blacksquare}$ holds and lastly it holds that $cH(\mathsf{C}[\mathsf{P}]) = cH(\mathsf{P})$ since the context C does not add any new $actor$s. We describe briefly what equalities are used in the steps indicated in Figure 10. In step (a) and (c) of Figure 10, we use the following identities for $\mathsf{P} = \mathsf{P}_1$ or $\mathsf{P} = \mathsf{P}_2$ respectively:

$$behav(\mathsf{C}[\mathsf{P}]) = behav([\![\mathsf{C}[\mathsf{P}]]\!])$$
$$[\![\mathsf{C}[\mathsf{P}]]\!] = [\![\mathsf{C}]\!]_{cH(\mathsf{P}),\blacksquare}\,[[\mathsf{P}]]$$

while step (b) relies on $[\![\mathsf{P}_1]\!] \simeq_{\mathbf{ctx}} [\![\mathsf{P}_2]\!]$ and $cH(\mathsf{P}_1) = cH(\mathsf{P}_2)$.

B. RrHC

To prove RrHC, we make use of a back-translation of contexts. The equivalence preservation of FAC ('$\Longleftarrow$') will also follow from the existence of the back-translation as shown later. A back-translation is a mapping from contexts in the target to contexts in the source. The back-translation should be defined such that a target context, $\mathbf{C}$, linked with a compiled program, $[\![\mathsf{P}]\!]$, results in the same evaluation as the back-translated context, $bcktr(\mathbf{C})$ linked with P. Our back-translation, $bcktr_{\mathbf{idT},\mathsf{k}}(\cdot)$, should only depend on $cH(\mathsf{P})$, and the term k which should neither be present in P nor in $\mathbf{C}$. The existence of such a back-translation then exactly shows the existence of the source context $\mathsf{C_S}$ from Theorem 1.

The construction of the back-translation is a particularly hard part of the proof because the back-translation needs to imitate certain behaviour of the target context without relying on actors since no new actors can be added to the context. This means that all authenticated communication that is not done by compiled versions of exact copies of actors in P should be imitated by communication from $anon$. To imitate this authenticated communication, we use anonymous communication over channels with specific names that incorporate every argument from $\overline{\mathbf{N}}^{\mathbf{auth}}\langle \mathbf{M}, \mathbf{a_1}, \mathbf{h_1}\rangle$ or $\mathbf{N}^{\mathbf{auth}}(\mathbf{M}, \mathbf{h_2}, \mathbf{a_2})$. Unfortunately, these channels do not offer the same flexibility as authenticated send and receive. More specifically, such a way of communicating requires the send/receive channels to have exactly the same name, whereas authenticated communication allows $\mathbf{a_1}$ and $attest(\mathbf{h_2})$ to differ when $\mathbf{h_2} = any$. This can be emulated by back-translating authenticated sends and receives not to one secure channel, but to a non-deterministic choice of both a channel with the correct attest or a channel with $anon$ instead of the attest. This way, a back-translation of functional communication will still be able to communicate in both ways by only using the exact equality of channel names.

As mentioned before, there are no attests in the source language, hence we use τ_{k} from Theorem 1 as a back-translation on the terms. Due to the mapping of terms and the fact that there is no exact bisimulation relation[4] as in the case of the correctness, there is no exact behavioural equality, but

$$behav(bcktr_{cH(\mathsf{P}),\mathsf{k}}(\mathbf{C})[\mathsf{P}]) = \tau_{\mathsf{k}}(behav(\mathbf{C}[\![\mathsf{P}]\!])) \qquad (1)$$

holds up to subtraces of silent steps of finite length, where $bcktr_{cH(\mathsf{P}),\mathsf{k}}(\mathbf{C})$ is the back-translation of the context assuming k does not appear in $\mathbf{C}$ and P.

The back-translation of the context is dependent on the identities table of the linked program P because the back-translation has to be able to identify the sends or receives that can communicate with compiled actors in $[\![\mathsf{P}]\!]$. If a $getAttest(\mathbf{x}, \mathbf{d}, \mathbf{D})$ is applied to a compiled program, then the back-translation is an actor containing exactly the corresponding source program. If a program requests to communicate with a program with a hash from the table but it was not $getAttest(\mathbf{x}, \mathbf{d}, \mathbf{D})$ of a compiled program, then we compile it to an anonymous communication with the right actor. If communication happens anonymously (i.e., $anon \rightarrow any$ in P), then back-translation should produce a process capable of communicating with an actor from an anonymous source.

We note that although the proof is somewhat complicated, we believe that the hardest and most interesting challenges are inherent to the problem and independent of the chosen model.

To understand this in more detail and see the technical execution of this argument, we refer the interested reader to the accompanying technical document.

[4]The back-translation introduces some extra silent steps, so a simulation relation with skipping [30] is used.

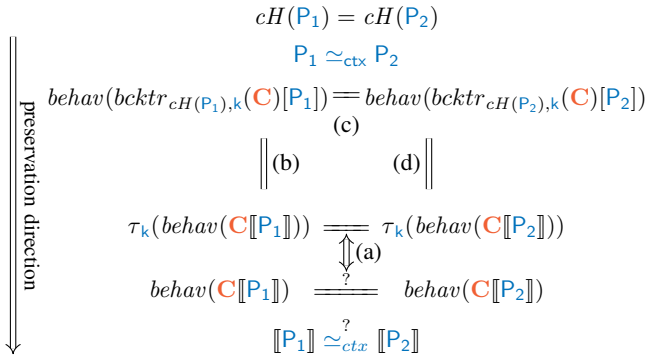

Fig. 11: Diagram of the proof of equivalence preservation of FAC. This diagram is adapted from the diagram in [29].

C. Equivalence preservation for FAC

Similarly to RrHC, also the equivalence preservation for FAC ('$\Longrightarrow$') is implied by Equation 1, following the reasoning shown in Figure 11. Below, the equalities used in each of the indicated steps in Figure 11 are briefly described. In step (a) of Figure 11, the injectivity of $\tau_{\mathsf{k}}(\cdot)$ is used where $\tau_{\mathsf{k}}(\cdot)$ is injective because k is taken such that $\mathbf{C}$, P_1 and P_2 do not contain the term k. In step (b) and (d) of Figure 11, Equation 1 is used respectively with $\mathsf{P} = \mathsf{P}_1$ and $\mathsf{P} = \mathsf{P}_2$. In (c) in Figure 11, the assumptions $\mathsf{P}_1 \simeq_{\mathsf{ctx}} \mathsf{P}_2$ and $cH(\mathsf{P}_1) = cH(\mathsf{P}_2)$ are used.

VI. Related work

In this section, we discuss work related to this paper. First, we discuss several models of RA that are meant to verify the correctness of RA implementations. Then, we discuss the verification of cryptographic protocols using RA. Subsequently, we discuss another paper that proposes solutions to obtain mutual authentication. Next, works that use the π-calculus and RA in the context of access control policies are discussed. Finally, we discuss actor frameworks that inspired π_{Actor} and we discuss other instances where secure compilation is used to prove the security of protocols given some primitives. There is also some research on verifying the correctness of distributed systems [31], [32], verified designs for RA [24] and the hardware security requirements of systems supporting RA [33], but we will not discuss it here. Furthermore, the paper that introduced the MAGE protocol [9] is discussed in Section II-c and will not be discussed again here.

Sardar, Musaev, and Fetzer [34] proposed a model to verify an implementation of RA written in ProVerif (which is based on the applied π-calculus) which is tailored to the TDX design based on Intel's official documentation. Their model contains several processes by default, such as the quoting enclave, the guest trust domain, the TDX module, and the CPU hardware. The model is also used to verify Intel's description of the TDX design. Examples of platform-independent frameworks to reason about implementations of RA are TAP [21] and the framework of Xu et al. [35]. Both frameworks are meant specifically for enclave platforms and both their abstractions are more low-level than those in our model. For example, TAP requires reasoning at assembly level and both frameworks assume an enclave memory access model, in contrast to our model that tries to abstract away these details. Subramanyan et al. [21] also showed that TAP supports secure remote execution and that formalisations of both Intel SGX and MIT Sanctum are refinements of TAP. The main goal of these models is to verify that RA works properly and is secure, whereas the goal of our model is to model protocols that use RA.

Next to models about implementations, there is also research into verifying protocols using RA, similarly to our verification of MAGE. One such example is the research by Barbosa et al. [36], which discusses attested computation, key exchange for attestation, and secure outsourced computation. They used probabilistic methods, such as security games, and they model the system at a lower level than our model, e.g., they initialise the hardware and explicitly build all attestation on top of digital signatures and secret information from the initialisation of this hardware. Fotiadis et al. [37] have also proposed a method using the SAPiC calculus of the Tamarin prover (akin to the applied π-calculus) to model RA. The paper presents a method to verify protocols using RA and shows one example instantiating the method for the TPM, by adding idealised versions of each of the commands in the TPM that are integral to support RA. They also assume an S-RTM and establish a chain of trust.

Another solution for mutual attestation was proposed by Zheng and Arden [12] with their Decent framework. In this work, mutual attestation is also resolved by preprocessing a list of authenticated enclaves and comparing the list at runtime before communicating. This is very similar to MAGE, but they do not use the properties of the hash resulting in a somewhat more complicated communication. Orthogonally to the mutual attestation solution they also provide a solution to dynamically authenticate and revoke enclaves and to delegate authentication for performance reasons. Specifically, to delegate authentication they let enclaves create attested tokens using RA and using these tokens they can sign digests of other components that they in turn checked using local attestation. This signed digest can then be used as an attest for that component. Verification can similarly be obtained. These things can be modelled in $\pi_{\mathbf{RA}}$, but for example to model the generation of the tokens we need the functions, $package(\mathbf{M}, \mathbf{b})$ and $check(\mathbf{pa}, \mathbf{h}, \mathbf{b})$, mentioned in Section III-C and public key cryptography techniques. This is just an extension of the signature of $\pi_{\mathbf{RA}}$. Additionally, Zheng and Arden [12] proved the security and authenticity of the Decent framework in the case of communication between enclaves from the list of authenticated components and the case of communication between enclaves verified by the verifier. The first of these proofs overlaps with our proof. However, the proof only applies to a specific case and any compositionality can only be assumed by the simplicity of the studied case. In contrast, our proof shows the compositionality for any program by using secure compilation which uses similar reasoning to the

real/ideal framework [13], [14]. We also show more than secrecy and authenticity as any security property (relational hyperproperty) that holds for programs of π_{Actor} (or at least for any given set of actors) also holds for their compilations.

Two frameworks supporting RA based on the π-calculus are presented in the extended abstract by Pitcher and Riely [38] and the paper by Cirillo and Riely [39]. Contrary to our model, these frameworks deal with S-RTM, thus putting a lot of effort into maintaining the chain of trust. Moreover, in the paper by Cirillo and Riely, the framework is an extension of a higher-order π-calculus, i.e., a version of the π-calculus where terms can contain programs. These frameworks [38], [39] are both typed and the type system is used to specify access policies based on the remote attestation primitive. In these frameworks one cannot use attestation to identify a program, only to prove that it satisfies the type system.

Our language π_{Actor} with actors is inspired by the EActors framework [25]. The EActors framework is specifically designed for Intel SGX and it is not a formal framework, but a high-level language aiming to make programming with remote attestation easier. Noorman, Mühlberg and Piessens [26] discuss a framework of modules that are all event-handlers with fixed input and output channels and this is fairly similar to a language using actors. In their supplementary material, they also prove secure compilation from this language with modules to an enclave architecture that uses attestation in a way inspired by Sancus [3] and they solve circularity by relying on a trusted third party.

Some works have used secure compilation to prove the security of certain primitives. For instance, Abadi, Fournet, and Gonthier [27] used two variants of the join-calculus, a process calculus, and showed secure compilation between the two. This secure compilation showed how authenticated communication can be provided by using constructs from cryptography. Lastly, Laud [40] established a secure compilation result to show that asynchronous method calls and futures can be built-up from explicit communication primitives and cryptographic operations.

VII. Conclusion

This paper presented a novel model for systems supporting remote attestation that is platform-independent and sufficiently abstract to allow modelling protocols relying on RA without coupling to a specific memory protection model or the use of a low-level language. This model is a variant of the applied π-calculus and is called $\pi_{\mathbf{RA}}$.

Moreover, we have used $\pi_{\mathbf{RA}}$ to formally verify the security of the MAGE protocol [9] which offers a solution for circular references in mutual attestation. The security of MAGE is shown by a secure compilation result from a source language (π_{Actor}) that uses named actors and secure communication based on these names, to the target language, $\pi_{\mathbf{RA}}$. Lastly, the proof of secure compilation and related work is presented.

References

[1] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in 25th USENIX Security Symposium (USENIX Security 16). Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for cpu based attestation and sealing," Intel Corporation, Tech. Rep., August 2013. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/innovative-technology-for-cpu-based-attestation-and-sealing.html

[3] J. Noorman, J. Bulck, J. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for iot devices," ACM transactions on privacy and security, vol. 20, no. 3, pp. 1–33, 2017.

[4] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: a security architecture for tiny embedded devices," in Proceedings of the Ninth European Conference on Computer Systems, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, Apr. 2014, pp. 1–14. [Online]. Available: https://doi.org/10.1145/2592798.2592824

[5] "Trusted Platform Module Library Part 1: Architecture," Trusted Computing Group, Tech. Rep., 11 2019.

[6] S. F. J. J. Ankergård, E. Dushku, and N. Dragoni, "State-of-the-art software-based remote attestation: Opportunities and open issues for internet of things," Sensors, vol. 21, no. 5, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/5/1598

[7] "secureworker," 2019. [Online]. Available: https://www.npmjs.com/package/secureworker

[8] M. Abadi, B. Blanchet, and C. Fournet, "The applied pi calculus: Mobile values, new names, and secure communication," Journal of the ACM, vol. 65, no. 1, pp. 1–41, 2018.

[9] G. Chen and Y. Zhang, "MAGE: Mutual attestation for a group of enclaves without trusted third parties," in 31st USENIX Security Symposium (USENIX Security 22). Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-guoxing

[10] M. Patrignani, "Why should anyone use colours? or, syntax highlighting beyond code snippets," CoRR, vol. abs/2001.11334, 2020. [Online]. Available: https://arxiv.org/abs/2001.11334

[11] "TCG PC Client Specific Implementation Specification for Conventional BIOS," Trusted Computing Group, Tech. Rep., 02 2012.

[12] H. Zheng and O. Arden, "Secure distributed applications the decent way," in Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems, ser. ASSS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 29–42. [Online]. Available: https://doi.org/10.1145/3457340.3458304

[13] Y. Lindell, "How to Simulate It – A Tutorial on the Simulation Proof Technique," in Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich, Y. Lindell, Ed. Cham: Springer International Publishing, 2017, pp. 277–346. [Online]. Available: https://doi.org/10.1007/978-3-319-57048-8_6

[14] R. Canetti, "Universally composable security: a new paradigm for cryptographic protocols," in Proceedings 42nd IEEE Symposium on Foundations of Computer Science, 2001, pp. 136–145.

[15] M. Abadi, "Protection in programming-language translations," in Secure Internet Programming, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, vol. 1603, pp. 19–34.

[16] M. Patrignani, A. Ahmed, and D. Clarke, "Formal approaches to secure compilation: A survey of fully abstract compilation and related work," ACM Comput. Surv., vol. 51, no. 6, feb 2019. [Online]. Available: https://doi.org/10.1145/3280984

[17] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in 2019 IEEE 32nd Computer Security Foundations Symposium (CSF), 2019, pp. 256–25 615.

[18] K. Vrancken, F. Piessens, and R. Strackx, "Securely deploying distributed computation systems on peer-to-peer networks," in ACM/SIGAPP Symposium on Applied Computing. Association for Computing Machinery, Apr. 2019, pp. 328–337.

[19] N. Smart, Cryptography Made Simple, 1st ed., ser. Information Security and Cryptography. Cham: Springer International Publishing : Imprint: Springer, 2016.

[20] V. Costan and S. Devadas, "Intel SGX explained," IACR Cryptol. ePrint Arch., p. 86, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[21] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2435–2450. [Online]. Available: https://doi.org/10.1145/3133956.3134098

[22] "Secure Encrypted Virtualization API Version 0.24," Advanced Micro Devices, Inc., Tech. Rep., 04 2020.

[23] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in Proceedings of the 26th Symposium on Operating Systems Principles, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 287–305. [Online]. Available: http://doi.org/10.1145/3132747.3132782

[24] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "Vrased: A verified hardware/software co-design for remote attestation," in Proceedings of the 28th USENIX Conference on Security Symposium, ser. SEC'19. USA: USENIX Association, 2019, p. 1429–1446.

[25] V. Sartakov, S. Brenner, S. Ben Mokhtar, S. Bouchenak, G. Thomas, and R. Kapitza, "Eactors: Fast and flexible trusted computing using sgx," in Proceedings of the 19th International Middleware Conference, ser. Middleware '18. ACM, 2018, pp. 187–200.

[26] J. Noorman, T. Mühlberg, and F. Piessens, "Authentic execution of distributed event-driven applications with a small tcb," vol. 10547, Livraga, G. Springer; Heidelberg, DE, 2017, pp. 55–71. [Online]. Available: https://doi.org/10.1007/978-3-319-68063-7

[27] M. Abadi, C. Fournet, and G. Gonthier, "Authentication primitives and their compilation," in Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 302–315. [Online]. Available: https://doi-org.kuleuven.e-bronnen.be/10.1145/325694.325734

[28] D. Apon, D. Dachman-Soled, H. Gong, and J. Katz, "Constant-Round Group Key Exchange from the Ring-LWE Assumption," in Post-Quantum Cryptography, J. Ding and R. Steinwandt, Eds. Cham: Springer International Publishing, 2019, pp. 189–205.

[29] M. Patrignani, E. M. Martin, and D. Devriese, "On the semantic expressiveness of recursive types," Proc. ACM Program. Lang., vol. 5, no. POPL, jan 2021. [Online]. Available: https://doi.org/10.1145/3434302

[30] A. Chlipala, "Formal reasoning about programs," http://adam.chlipala.net/frap/frap_book.pdf, 2015.

[31] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," SIGPLAN Not., vol. 50, no. 6, p. 357–368, jun 2015. [Online]. Available: https://doi-org.kuleuven.e-bronnen.be/10.1145/2813885.2737958

[32] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: Proving practical distributed systems correct," in Proceedings of the 25th Symposium on Operating Systems Principles, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–17. [Online]. Available: https://doi-org.kuleuven.e-bronnen.be/10.1145/2815400.2815428

[33] G. Cabodi, P. Camurati, C. Loiacono, G. Pipitone, F. Savarese, and D. Vendraminetto, "Formal Verification of Embedded Systems for Remote Attestation," vol. 14, p. 10, 2015.

[34] M. U. Sardar, S. Musaev, and C. Fetzer, "Demystifying attestation in intel trust domain extensions via formal verification," IEEE Access, vol. 9, pp. 83 067–83 079, 2021.

[35] S. Xu, Y. Zhao, Z. Ren, L. Wu, Y. Tong, and H. Zhang, "A Symbolic Model for Systematically Analyzing TEE-Based Protocols," in Information and Communications Security, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, vol. 12282, pp. 126–144, iSSN: 0302-9743.

[36] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi, "Foundations of hardware-based attested computation and application to sgx," in 2016 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2016, pp. 245–260.

[37] G. Fotiadis, J. Moreira, T. Giannetsos, L. Chen, P. B. Rønne, M. D. Ryan, and P. Y. A. Ryan, "Root-of-Trust Abstractions for Symbolic Analysis: Application to Attestation Protocols," in Security and Trust Management, R. Roman and J. Zhou, Eds. Cham: Springer International Publishing, 2021, pp. 163–184.

[38] C. Pitcher and J. Riely, "Dynamic Policy Discovery with Remote Attestation," in Foundations of Software Science and Computation Structures, L. Aceto and A. Ingólfsdóttir, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 111–125.

[39] A. Cirillo and J. Riely, "Access control based on code identity for open distributed systems," in Lecture Notes in Computer Science, ser. Lecture Notes in Computer Science, vol. 4912. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 169–185.

[40] P. Laud, "Secure Implementation of Asynchronous Method Calls and Futures," in Trusted Systems, C. J. Mitchell and A. Tomlinson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 25–47.