POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Formal verification of security properties for remote attestation protocols

Supervisors
prof. Riccardo Sisto
prof. Fulvio Valenza
dott. Simone Bussa

Candidate
Alessandro Di Lorenzo

Academic Year 2021-2022

Summary

Formal verification refers to a set of techniques based on formal methods, which aim to assess whether a certain formal model is well-defined or that a system satisfies some specific properties. This kind of formal analysis lends itself very well to the use of automated algorithms that help reach the desired outcome in reasonable time. This technique has been successfully applied in the verification of cryptographic protocols in the literature, to evaluate their security in case of hostile actors interference; the main advantage of this analysis is its ability to identify potential vulnerabilities that would not be intuitively detectable, as was the case for widely used protocols which were believed secure for years.

A possible application of cryptographic protocols is to protect the process of Remote Attestation, which is a key component of a trusted computing environment. In particular, the attestation purpose is to assess whether a certain system, that wishes to join a trusted network, is not behaving maliciously. When the attestation process is carried out through a public network, it is necessary to employ a number of security measures that can prevent adversaries from illegitimately obtaining a trusted state.

Some automatic formal verification techniques, through the use of the popular and successful tool *ProVerif* and its extension, are applied on two main security protocol for Remote Attestation procedures developed in the context of a trusted fog computing platform: a simpler implementation that is proved to have some flaws in its definition and a more complex one, as an enhanced version of the previous, fixing most of the initial issues. The results obtained from the analysis are only partial with respect to the initial objective to prove all properties on both protocols, since the tool used is shown to not be the ideal choice for this particular application.

Contents

List of Figures 4

1 Introduction 5
- 1.1 Thesis introduction . . . 5
- 1.2 Thesis description . . . 6

2 Formal verification of cryptographic protocols 7
- 2.1 ProVerif . . . 8
  - 2.1.1 ProVerif limitations and global state extensions . . . 9

3 Remote attestation in a trusted fog computing platform 11
- 3.1 Remote attestation . . . 11
- 3.2 The RAINBOW project . . . 13

4 Analysed protocols 15
- 4.1 Attestation by quote . . . 16
  - 4.1.1 System model . . . 16
  - 4.1.2 Request for measurement update . . . 16
  - 4.1.3 Attestation phase . . . 18
- 4.2 Oblivious remote attestation . . . 19
  - 4.2.1 System model . . . 20
  - 4.2.2 AK creation request . . . 22
  - 4.2.3 PCR management . . . 23
  - 4.2.4 Measurement update . . . 26
  - 4.2.5 Oblivious remote attestation phase . . . 28

5 Thesis objective 32

6 Formal modelling of analysed protocols 34
- 6.1 Attestation by quote . . . 34

6.1.1 Simple attestation phase . . . 34
6.1.2 PCR update request and attestation . . . 36
6.1.3 Multiple fog nodes involvement . . . 37
6.1.4 Fog node and TPM as separate entities . . . 38
6.1.5 Verification of RAINBOW security properties . . . 40
6.2 Oblivious remote attestation . . . 42
6.2.1 Attestation Key creation . . . 42
6.2.2 PCR or NV-PCR attachment . . . 46
6.2.3 PCR or NV-PCR detachment . . . 48
6.2.4 Measurement update . . . 51
6.2.5 ORA . . . 54
6.3 Variations with global state handling . . . 57
6.3.1 Attestation by quote . . . 57
6.3.2 Oblivious remote attestation . . . 59

7 Formal verification results 61
7.1 Attestation by quote . . . 61
7.2 Oblivious remote attestation . . . 63
7.3 Assessment on security properties . . . 66

8 Conclusions 67

Bibliography 69

A Complete formal models implementation 71
A.1 Attestation By Quote . . . 71
A.2 Oblivious remote attestation . . . 75
A.2.1 AK creation . . . 75
A.2.2 NV-PCR attach . . . 79
A.2.3 NV-PCR detach . . . 82
A.2.4 Measurement update request . . . 87
A.2.5 ORA . . . 91

List of Figures

3.1 Internal architecture of TPM 2.0 [20] 12

4.1 System model with orchestrator and managed *VFs* 17
4.2 Update measurements request 18
4.3 Attestation by quote protocol 18
4.4 Refined system model with initial knowledge 21
4.5 Complete workflow of the ORA protocol 22
4.6 AK creation phase 24
4.7 Attaching normal or NV-PCRs 25
4.8 Detailed steps of Algorithm 1 26
4.9 Detailed steps of Algorithm 2 27
4.10 Detailed steps of Algorithm 3 27
4.11 Detailed steps of Algorithm 4 28
4.12 Detachment of normal or NV-PCRs 29
4.13 Measurement update phase 30
4.14 Oblivious remote attestation phase 31

7.1 Attack trace showing the fake update request 62
7.2 Attack trace with fake tracer values and node trusted 63

Chapter 1

Introduction

1.1 Thesis introduction

In recent years, the ever-growing diffusion of smart devices has led to the integration of novel technologies in a wide variety of fields ranging from industry automation to connected cars to healthcare systems and many other aspects of our life. The concept of *Internet of Things* is used to describe this conjunction between the digital and the physical world, where objects equipped with processing hardware or sensors are able to connect and exchange data among themselves and over the Internet.

In this context, cloud-based technologies, which have also become increasingly popular in the last few years, represent an optimal method to manage the data collected by the above mentioned devices. This can be done with both a centralised approach, following the traditional cloud architecture where data is sent over the network and processed remotely, or with a decentralised one, shifting most of the computational work to devices which are closer to or part of the edge of the network. *Fog computing* is a paradigm that follows the latter approach and it is convenient in critical services where certain QoS requirements have to be fulfilled, specifically in terms of processing speed in order to achieve real-time reaction when needed, but also in terms of low bandwidth consumption.

It is clear that this kind of decentralised architecture needs particular care in handling its security in order to guarantee identity management, resource integrity and data protection. It is necessary to establish chains of trust between all participating entities in the network, such that every node can be securely identified and managed. This can be achieved through *remote attestation* schemes, ensuring that a certain initially untrustworthy entity can be safely declared as trusted, that

is it is not carrying out any adversarial activity, and included in the chain of trust.

Remote attestation can be achieved through different means by leveraging both hardware and software components. The schemes for remote attestation can be defined as security protocols whose specifications and implementation have to be considered carefully since they can easily lead to errors, which, if left undetected, can compromise the whole system they are applied to. Therefore, modelling and verification of security protocols through formal methods is a technique often employed to identify potential errors and vulnerabilities and to test whether the behaviour is the expected one in the presence of adversaries.

1.2 Thesis description

The following part of this thesis will be structured as described below

Chapter 2. Introduces the formal verification techniques and tools which will be used in this work.

Chapter 3. Introduces the concept of remote attestation and its application in the context of a trusted fog computing environment.

Chapter 4. Describes the design and functionalities of the protocols formally verified in this work.

Chapter 5. Presents the objective of this thesis and the security properties that need to be verified.

Chapter 6. Reports the implementation of the protocols as formal models to be verified.

Chapter 7. Reports the results obtained by the automatic formal verification carried out on the models described in the previous chapter.

Chapter 8. Conclusions.

Appendix A. Complete copy of the formal models developed.

Chapter 2

Formal verification of cryptographic protocols

A cryptographic protocol is designed in order to satisfy specific security requirements. When the protocol is not designed correctly, however, it may fail its purpose, thus exposing the system it was meant to protect to external interference by possibly malicious actors. Nevertheless, carefully analysing a protocol by manual review can often not be enough to identify potential flaws in the design. An example for such flaws, which were discovered much later on widely used and presumably secure protocols, is given by the *Needham–Schroeder* case; both versions of the protocol were found to be flawed: the symmetric key one allowed the use of a compromised old session key, as discovered by Denning and Sacco [10], whereas the public key authentication one allowed a man-in-the-middle attack, as described by Lowe [13]. An appropriate solution to identify potential problems, which could otherwise go undetected, is the use of formal methods. Once a formal model, that is a formal mathematical abstraction, of a system has been specified, formal verification techniques can be applied in order to assess whether some requirements are satisfied. The latter are properties usually specified in a logic language which can be verified with two possible methods: *model checking* and *theorem proving*. Given a model $\mathbf{M}$ and a property $\mathbf{f}$, model checking verifies whether $\mathbf{f}$ is true for $\mathbf{M}$, or otherwise provides a counter-example as proof of non-validity. Instead, given a theory $\mathbf{T}$ and a property $\mathbf{f}$, theorem proving aims to verify the validity of $\mathbf{f}$ for all possible interpretations of $\mathbf{T}$ by providing a proof; if a proof is not found, nothing can be known.

When handling cryptographic protocols, there are various scenarios that have to be taken into account, such as concurrent sessions or the unpredictability of

attacker behaviour. It is therefore necessary to define the scope of what an attacker can do and when a protocol has to considered compromised. To this purpose, two main approaches are reported below.

Symbolic models. Based on high-level abstractions; one notable example is the *Dolev-Yao* model [11], which represents the network as a set of honest users exchanging messages, consisting of symbolic terms, and which considers ideal cryptographic operations, such that no attacks can be based on weak cryptography exploitation. Moreover, the attacker can intercept all traffic traversing the network, modify or delete messages, create new ones and also use every cryptographic operation which is available to honest users. However, terms which are specified as secret cannot be guessed or obtained by the attacker, at least initially. Standard security properties like authentication, secrecy and integrity can be verified by using both *model checking* or automated *theorem proving*.

Computational models. More complex approach, since it uses a low-level representation, but for this reason it is also potentially more accurate. Data is modelled as bitstrings and cryptographic operations as algorithms, whereas the attacker is considered any polynomial-time algorithm. The analysis is probabilistic, which means that every operation is considered possible but some are associated to a very negligible chance of being executed. The objective is to prove that no attacker can reach a given goal in polynomial time with non-negligible probability.

Both approaches do not consider side-channel attacks, such as those based on timing.

2.1 ProVerif

The tool used to carry out the verification of the protocols described in this thesis is *ProVerif* [5], which is an automatic theorem prover based on the Dolev-Yao formal model. The protocol formal specification is expressed by an extended version of the *Applied Pi-Calculus* [1][2] which supports types, along with some queries representing the security properties to be verified. Both are then processed and translated into a set of *Horn clauses* to which a resolution algorithm is applied in order to find a proof for the requested properties. If a proof is not found, then the tool tries to reconstruct an attack, in the form of an execution trace which falsifies the desired property. However, this process may yield false attacks and it is also not exhaustive, that is, even if an attack was not found, it does not mean that no attack is possible. *ProVerif* is able to prove reachability properties (e.g. secrecy of some term), correspondence assertions (e.g. authentication or more generally

correspondence between events) and observational equivalence (e.g. strong secrecy, offline-guessing attacks or equivalence between processes that only differ by terms).

The pi-calculus used to specify the protocol is meant to represent concurrent processes which interact through communication channels such as the Internet; each honest actor in the protocol is modelled by a calculus process and there is no need to model an attacker, since its behaviour is considered unpredictable. *ProVerif* also supports unbounded parallel sessions of the protocol and the use of a wide range of cryptographic primitives, including but not limited to symmetric and asymmetric encryption, digital signatures, hashing, expressed through a set of rewrite rules or equations.

2.1.1 ProVerif limitations and global state extensions

ProVerif has been successfully used to prove the correctness and security of several protocols from the literature, such as the above-mentioned Needham-Schroeder protocol; additionally, some notable example of case studies include the handover procedures between LTE-LTE and LTE-UMTS cells [9], the authentication protocols in Trusted Platform Modules [7], the secure messaging protocol of the *Signal* app [12] and *TLS 1.3 Draft-18* [4]. Nevertheless, there are protocols, particularly those relating to hardware devices such as TPMs or smart cards, which may need to deal with information that should be made available to all sessions of the processes engaging in communication. Such kind of information cannot be handled locally, but rather it should be modelled as a global state shared among different sessions, potentially modifiable an unspecified number of times. Some protocols may need the use of memory cells to represent the internal storage of hardware devices, such as those holding cryptographic keys, as well as persistent databases or counters.

The standard implementation of *ProVerif* does not explicitly support this kind of functionality. A possible workaround to solve this issue is declaring a `free` name, which is akin to a global variable in standard programming languages, of type **`channel`**; this kind of variable allows for input and output by processes and is normally intended for message exchange. Since `free` names are shared among processes, a channel could be used as an abstraction of a global state with processes reading and writing on it as if it was actual memory. A channel used in this manner should be declared as `private`, since all free names are known also to the attacker by default. However, this solution is not optimal as it would often still yield false attacks or not provide any result. That is because, due to its internal abstractions into *Horn clauses*, *ProVerif* introduces over-approximations that fail to handle global states, as values inputted into a channel are persistent and not overwritten whenever a new value is saved, meaning that the tool only guarantees that the read value was previously written on the cell but not necessarily that it was the

last one. This issue is well known and a certain number of alternatives have been developed in order to overcome it. The work presented in this thesis will employ two of these solutions.

StatVerif. [3] Stand-alone extension of *ProVerif* enriching its process calculus. Introduces new constructs in the language to explicitly handle global states and therefore allows for correct translation into Horn clauses that avoid the above-mentioned false attacks. In particular, it introduces the new type keyword `cell` which models a global variable that allows multiple read and write operations, which replace the old value, and also locking and unlocking to solve synchronisation issues with concurrent processes. However, while the number of rewrites is theoretically unlimited, multiple value updates may largely increase the complexity of the analysis leading very easily into state explosion and therefore no termination in reasonable time.

GSVerif. [8] Acts as a front-end to be used in combination with standard *ProVerif*, processing files written in the same specification language, aptly modified with specific keywords, in order to produce a new file automatically annotated with events that record channel usage, values and possibly freshness indicators. It introduces specific keywords, to be used in the case of channels employed as global states, as in the standard workaround, such as `cell`, `counter`, `uniqueAction`, `uniqueComm` which all model a specific use case. Moreover, the keyword `precise` is also available, which tries to automatically find the best option, relatively to the implementation in the existing processes.

Chapter 3

Remote attestation in a trusted fog computing platform

3.1 Remote attestation

In the context of a trusted computing environment, one of the key principles is the verification of consistent behaviour of communicating entities in order to achieve unambiguous trust between them, as defined by the TCG [19]. The process through which such trust is obtained is named *Attestation* and it is performed by having an initially untrusted entity, a *Prover*, report its current integrity state to an external trusted entity, the *Verifier* or *Requestor*, which can assess whether the state of the *Prover* matches the expected one. Whenever this process implies interaction through network protocols, the term ***Remote Attestation*** is employed.

The main idea behind the concept of trusted computing is to have a specific component acting as *Root-of-Trust* (RoT), that is a piece of software or hardware which is inherently considered as trusted, thus serving as the starting point for the chain of trust. A *Trusted Platform Module* (TPM) whose internal architecture in shown in Figure 3.1, following the specifications by the TCG [21], is frequently chosen as RoT of a system. In its most secure implementation, a TPM is a dedicated tamper-resistant hardware cryptographic processor, able to handle a wide range of cryptographic primitives and algorithms, with secure internal storage for cryptographic keys and integrity measurement of system components.

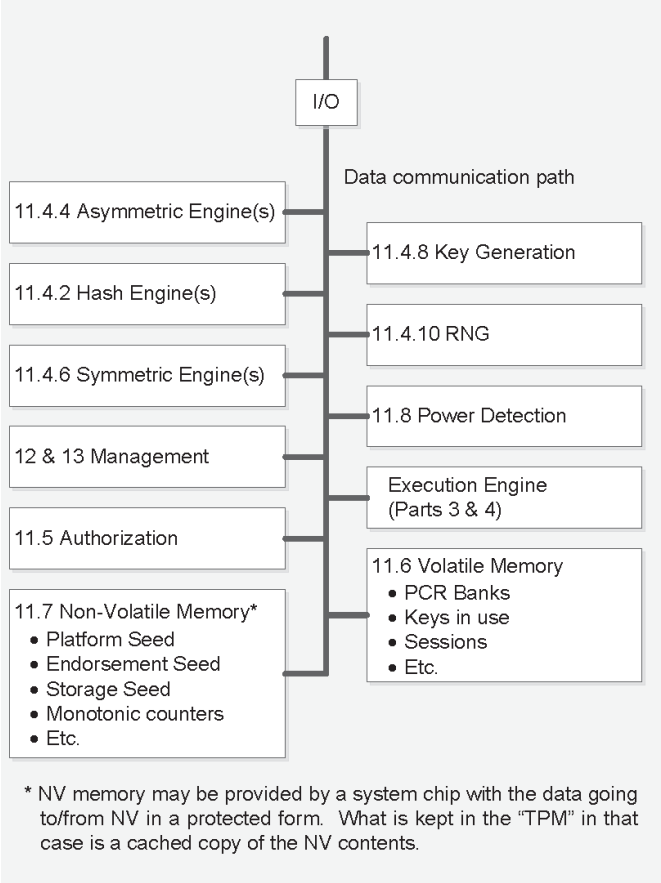

Figure 3.1. Internal architecture of TPM 2.0 [20]

The process of remote attestation, in particular, usually leverages on the integrity measurements stored in specific registers named *Platform Configuration Registers* (PCR). These registers are used to accumulate values obtained by the concatenation of the hashes of the *Trusted Computing Base* (TCB), that is the set of all software and/or hardware components that are considered critical to the security of the whole system. In particular, the PCR extension operation is carried out as below

$$PCR_{new} := H(PCR_{old}||h_{update})$$

Where H stands for the hashing operation with the chosen algorithm and h_{update} is the new digest computed over the latest measured component. Typically, at the start of the measurement process the PCRs are initialised to zero. The final value obtained represents the current integrity state of the system and it can be used as proof of its correct behaviour. One of the simplest techniques for remote

attestation would be having the *Prover* send a digital signature over the PCR values to the *Verifier*, which could in turn both verify the identity of the sender and whether the values received are equal to those computed as the expected ones.

3.2 The RAINBOW project

The remote attestation protocols described in this thesis are developed within the RAINBOW project [15], whose aim is to create a platform for trusted fog computing which enables easy deployment and management of different, scalable and secure IoT services, in the context of the ever-increasing diffusion of smart devices, drones, connected vehicles and systems in Industry 4.0 and the need to quickly process the data produced by shifting the main computational power from a centralised cloud infrastructure to a distributed edge one. As reported on the project official site [14] its main objectives are

Objective I. Provide an open and trusted fog computing reference architecture and highly relevant industry use-cases that facilitate the design, development and orchestration of scalable, heterogeneous, secure and privacy-preserving IoT services and cross-cloud applications, incorporating technological and business requirements coming from the industry, the research and academic community.

Objective II. Provide a set of innovative mechanisms and middleware tools for IoT orchestration, data collection and decentralised analytics that guarantees network security, data protection, identity management and resource integrity. The key characteristic of the middleware will be the embedded intelligence and remote attestation mechanisms for establishing trust and QoS requirements while coping with performance and network uncertainties.

Objective III. Enable secure and efficient data storage and processing at the fog and edge layer and facilitate the extraction of high-level analytic insights by introducing novel decentralised algorithms and open APIs.

Objective IV. Prove the applicability, usability, effectiveness of the RAINBOW integrated framework, models and mechanisms in industrial, real-life trustworthy services, applications and standards demonstrating and stress-testing the RAINBOW artefacts, methodologies and services under pragmatic conditions against a predefined set of use cases.

Objective V. Ensure wide communication and scientific dissemination of the innovative RAINBOW results to the industry, research and international community, to realise exploitation and business planning of the RAINBOW design models, software kits and orchestration mechanisms, to identify end-users and

potential customers, as well as to contribute specific project results to relevant open source communities.

Chapter 4

Analysed protocols

As per *objective II* of the RAINBOW project reported in Chapter 3, it is necessary to enable the secure and privacy-preserving enrolment into the Chain-of-Trust of fog/edge devices in order to establish trust-aware *Service Graph Chains* (SGCs), such that all communications from edge devices to fog nodes and backend cloud systems must support secure interactions between all participating entities. Some of the fundamental security design principles chosen to create such a secure environment contemplate

Root-of-Trust. TPM 2.0, following the official standard specification by the TCG [21], is the designated trusted component. A hardware TPM shall be used whenever the platform provides physical access, otherwise a software one must be implemented in a trusted environment.

Key management. Each key should be used for one single purpose, and all of them are to be securely stored and maintained by the TPM.

RAINBOW orchestrator. An orchestrator entity ($\mathcal{O}\texttt{rc}$) will be in charge of providing and updating policies on privacy and anonymity, managing the distribution of certificates and handling of requests for measurement updates, key creation and initial trust assessment.

Following this principles, the focus is on the provision of zero-touch configuration capabilities (fog nodes, wishing to join a fog cluster, adhere to the compiled attestation policies by providing verifiable evidence on their configuration integrity and correctness) with particular attention in preserving the privacy and anonymity of all participating nodes. The attestation protocols devised within the project and verified in this thesis will be described below.

4.1 Attestation by quote

4.1.1 System model

The design of this protocol is described in public deliverable D2.2 [17] of the RAINBOW project. The system model considered to enforce this protocol consists of a set of *Virtual Function* (*VF*) instances spawned and controlled by the $\mathcal{O}$rc as part of different service chains. It is assumed that each *VF* is a containerised micro-service, exploiting light-weight virtualisation techniques and therefore achieving high-level scalability and agility, while executing in an independent virtual environment equipped with a TPM, serving as RoT of the system. Furthermore, each *VF* has an associated configuration which is defined as the set of all objects (i.e. binary data) that are accessible through unique file identifiers and which are part of its TCB, therefore the measuring results of all of these objects are securely stored in the PCRs of the TPM. In formal terms the system can be described as

$$SG = \{s_1, s_2, ..., s_n\}, n \in \mathbb{N}^*$$

Where ***SG*** represents the Service Forwarding Graph maintained by the $\mathcal{O}$rc, which includes all service chains comprising of the *VFs*, such that

$$s_i = \{vf_1, vf_2, ..., vf_m\}, m \in \mathbb{N}^*, s_i \in SG$$

Additionally, each *VF* can be represented in its initial form by the following tuple

$$vf_i = (id, vPCR, state, EK_{pub}, AK_{pub})$$

where *id* is the identifier number of the *VF*, *vPCR* represents a set of mock *PCRs* whose measurement is computed according to the current deployed policy and which should match that of the actual PCRs it refers to if everything is behaving correctly, *state* refers to the current trust state of the *VF* and EK_{pub} and AK_{pub} respectively refer to the public parts of the Endorsement Key and the Attestation Key of the vTPM associated to vf_i. In order to provide real time measurement during system execution without the need for a reboot in case of changes, each *VF* is also equipped with a Runtime Tracer component ($\mathcal{T}$rce), which can record the state of binary data and securely store into the PCRs of the TPM. By definition, given an object identifier, the $\mathcal{T}$rce utility returns its corresponding binary data. A graphical representation of the described system is shown in Figure 4.1.

4.1.2 Request for measurement update

With the purpose of being able to manage changes in the configuration during the whole life cycle of a fog node, participating in the secure and trusted network,

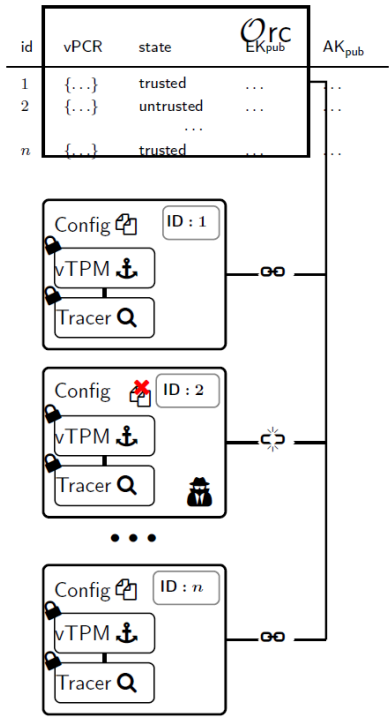

Figure 4.1. System model with orchestrator and managed *VFs*

it is necessary for the $\mathcal{O}$rc to notify all attested entities whenever new security attestation policies are enforced (e.g. when new vulnerabilities are identified). In such cases, the orchestrator autonomously computes the values relative to the expected change in configurations and accumulates them in the vPCR structures of the corresponding vTPMs of the affected *VFs*. Subsequently, the orchestrator requests to perform a measurement update to the actual nodes which the VFs represent, by sending them both the index i of the PCRs affected by the update and the identifier for the new configuration file(s) to be measured. When a node receives such a request, it invokes its Tracer component in order to obtain the actual new configuration data, computes its hash (the algorithm solely depends on implementation and therefore is not considered in this description) and then invokes its TPM with a PCR_Extend request, providing the new hashed value to be concatenated to the old one as shown in Section 3.1. This simple update protocol is shown in Figure 4.2.

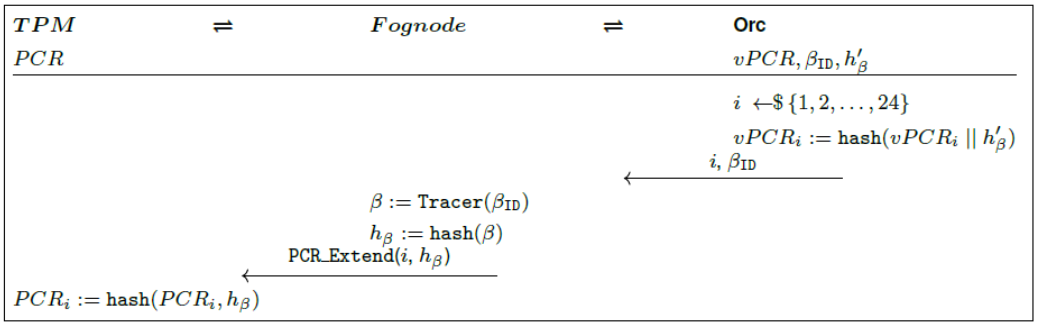

Figure 4.2. Update measurements request

4.1.3 Attestation phase

Once the new values have been correctly accumulated into the PCRs of the TPM of the devices involved, the orchestrator should verify whether this update was performed correctly and therefore evaluate the trustworthiness of the corresponding device. The $\mathcal{O}\mathtt{rc}$ is able to perform this evaluation because it actually knows what the correct values are as shown in the previous phase above. At this point, the orchestrator initiates the actual *Attestation by quote* protocol by sending both the index I of the PCRs which need to be checked and a nonce n, in order to ensure freshness and avoid replay attacks. Figure 4.3 shows all the steps of this protocol.

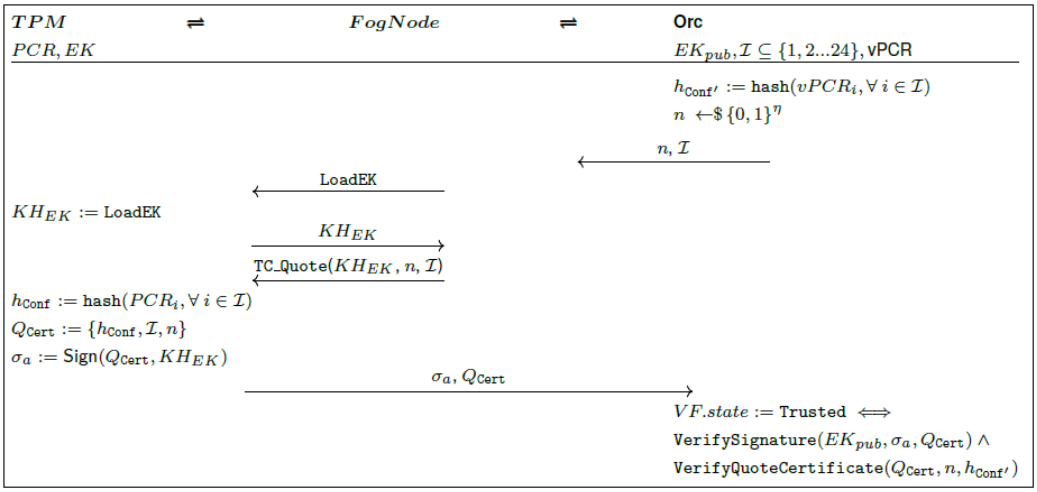

Figure 4.3. Attestation by quote protocol

It is assumed that the $\mathcal{O}$rc knows the correct hash of the PCRs to be measured and the public part of the Endorsement Key of the TPM hosted on the device of interest, since it is associated to the corresponding *VF*, as shown when describing the system model above.

Once the node receives the index of the PCRs to be evaluated and the nonce it invokes its TPM by using the `LoadEK` command which allows it to obtain the handle of the its Endorsement Key KH_{EK} and then uses the same handle, along with the nonce and the indexes, to request the creation of a Quote structure to the TPM, which in turn computes the value h_{Conf} which contains the hash of the PCRs at the specified indexes; the TPM then builds the Q_{Cert} object by combining the hashed value, the indexes and the nonce and finally computes a digital signature on it by using the private part of its *EK*. The signature and the Quote structure are eventually sent back to the orchestrator, which verifies the signature on the quote using the known EK_{pub} and whether all the other values (i.e. hash, indexes and nonce) match the correct ones. If the verification is successful the $\mathcal{O}$rc declares the Fog Node as *Trusted* by updating the state variable associated to its *VF*.

It is worth noting that this protocol can only attest integrity relatively to the last known measurement, therefore whenever the update measurement request protocol is executed, this protocol should be run in conjunction immediately after, in order to always assure run-time integrity.

4.2 Oblivious remote attestation

Apart from the *Attestation by quote* discussed above, Deliverable D2.2 [17] of the project also defines another attestation protocol, named *Attestation by proof* and whose analysis is not included in this work, with the objective of providing a way for *VFs* to verify trust among themselves and establish paths for the service chains in a distributed manner, without interacting with the central entity, represented by the $\mathcal{O}$rc. The main idea behind this protocol is to have the *VFs* create an asymmetric pair of *Attestation Keys* (*AK*), stored in the TPM, whose secret part usage is bound to the presence of specific $\mathcal{O}$rc-authorised values in a determined set of TPM PCRs. Once the *AK* is created and certified by the orchestrator, the *VF* it belongs to distributes its public part to all of its neighbours, namely all of the other *VFs* in the ***SG***, so that any *VF* that needs to verify the correct state of another, simply has to send a nonce as a challenge and verify that the signature computed over it is actually using the secret part of the *AK*.

This scheme for zero-touch configuration, however, is not optimal, as it presents three main issues

Static nature of *AK*. Each *AK* is actually bound to the single policy being

enforced when it was created. This implies that whenever a new policy is deployed a new *AK* must be created, verified, certified and distributed, therefore creating limitations on the efficiency of the whole RAINBOW attestation services.

Attestation Agent security. It is assumed that each measurement operation is carried out by the local $\mathcal{AA}\mathtt{gt}$ of the *VF*, not taking into account the possibility that the communication between the two parties may be compromised by an adversary, supplying incorrect data.

Exclusive use of normal PCRs. The number of PCRs inside a TPM is limited, and only a subset are *static*, that is they can't reset during run-time. Since each *VF* requires at least one associated static PCR, the number possible deployable *VFs* becomes limited.

All of these problems are addressed and resolved by the reinforced zero-touch configuration protocols presented in RAINBOW Public Deliverable D2.3 [18]. The solutions devised to solve the issues presented above include the use of repurposable *Attestation Keys*, *Attestation Agents* equipped with secure and unique hash keys and the use of the TPM non-volatile memory to create *static* NV-PCRs.

4.2.1 System model

The model considered for the enhanced version of the previous protocols is actually similar to the one presented in Section 4.1. The $\mathcal{O}\mathtt{rc}$ is spawning a set of containerised *VFs* instances as part of Service Graph chains. However, each *VF* is now equipped with three trusted components: a SW-TPM serving as RoT, a secure attestation agent and a secure tracer to measure its current configuration. A graphical representation of the system is shown in Figure 4.4. Each *VF*, in this case, has two sets of PCRs to separately track the the contents of the standards ones and of the ones created on non-volatile memory (NVPCRs). The *VF* initial knowledge also include three key handles: a storage key (SK) for the creation of AKs, its own SW-TPM endorsement key, which was negotiated with the orchestrator at the time of its creation, and a handle to the public of the $\mathcal{O}\mathtt{rc}$ own EK. Every attestation agent has also a unique key, shared with the orchestrator, in order to authenticate its involvement in the measuring process. This key is assumed to only be accessible by privileged code of the attestation agent itself. The $\mathcal{O}\mathtt{rc}$ maintains the $\boldsymbol{SG}$ with each spawned *VF*, to which it associates the public part of their endorsement key, the shared key with the attestation agent and both sets of mock PCRs and NVPCRs.

The protocol scheme, named *Oblivious Remote Attestation* (ORA), which leverages on this system model is basically an enhanced version of the *Attestation by*

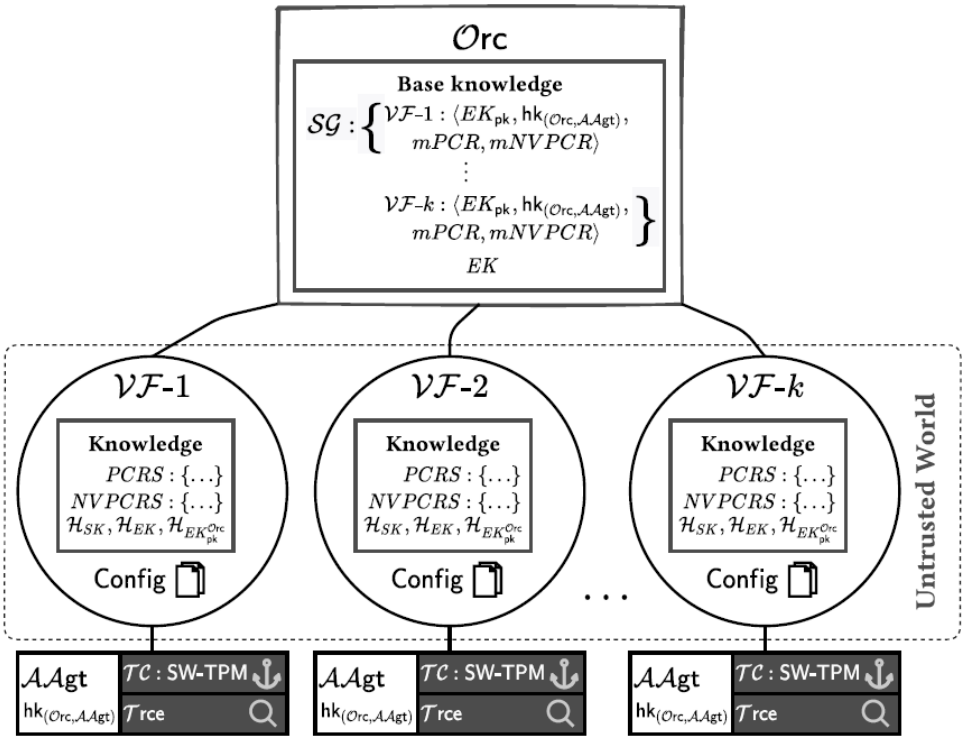

Figure 4.4. Refined system model with initial knowledge

proof protocol described above, such that each *VF* can have its correct configuration state attested by the $\mathcal{O}$rc and be allowed to use the created attestation key to attest its honest behaviour to any other *VFs*, without revealing any actual measurement information and therefore completely preserving privacy. While Figure 4.5 shows the complete workflow of this attestation scheme, the protocol is complex and comprising of various phases, which will be described in details in the following sections.

The protocol starts when a new *VF* requests to join the network. The $\mathcal{O}$rc requests the creation of an attestation key which will locked to a *flexible policy* bound to the endorsement key of the orchestrator itself, so that only the latter can actually authorise the use of the new attestation key. The $\mathcal{O}$rc verifies that the creation of the AK was performed correctly and then enrols the new *VF* into the service chain, while also advertising the public part of the new attestation key to all the neighbours in the chain. At this point, in order to allow the newly enlisted *VF* to prove its state, the $\mathcal{O}$rc sends to it a policy digest over its current acceptable configuration signed with its own private endorsement key. Whenever another

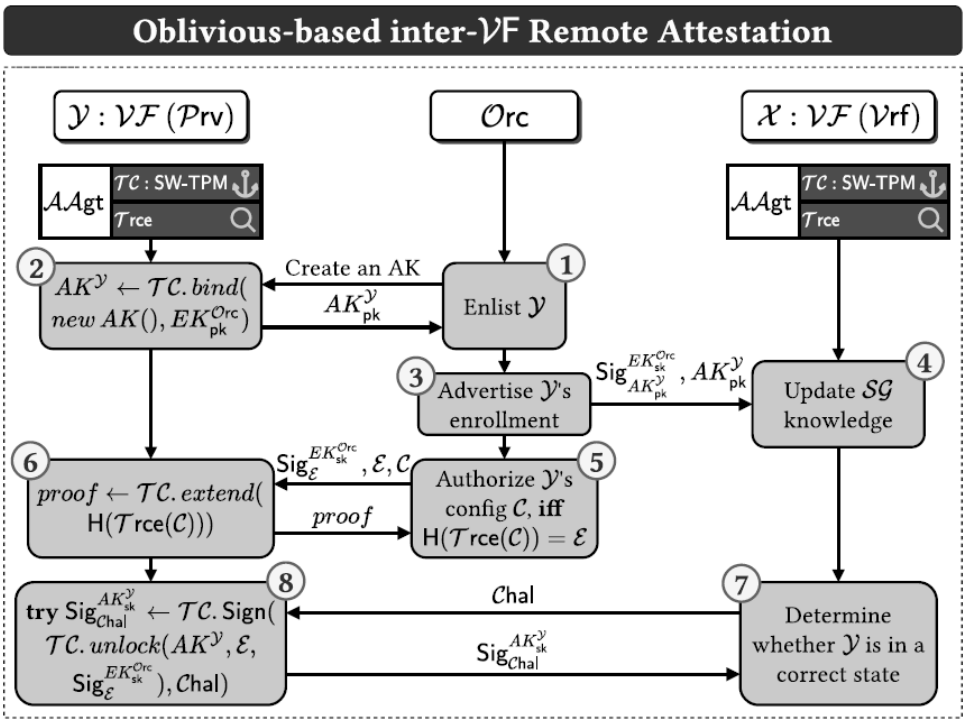

Figure 4.5. Complete workflow of the ORA protocol

VF from the same chain wants to assess the trustworthiness of the new one, it sends a nonce as challenge and expects to receive a signature with the previously advertised attestation key; the receiver of the challenge, measures its own state into its SW-TPM and if it matches the one authorised by the orchestrator, it is allowed to use its private attestation key to sign the challenge and thus proving its correct state.

4.2.2 AK creation request

At the beginning of the protocol, the `Orc` requests the creation of the attestation key. In particular, all of the steps of this phase are shown in Figure 4.6. The orchestrator computes a policy digest over the command code of `TPM2_PolicyAuthorize` and the name of its own endorsement key and sends it to the *VF* along with the template specifications of the key to be created. The policy computed is *flexible*, which means that any object bound to it can only be used inside the SW-TPM when some policy authorised by its owner is fulfilled. The *VF* then invokes its SW-TPM to create the AK requested, which is then returned along with a signed ticket providing information on its actual creation inside the TPM. After that, the

VF uses the newly received AK, its own EK and the creation ticket to invoke the `TPM2_CertifyCreation` command, such that the TPM provides proof of creating the AK by signing it along with other internal state information. At this point, the new AK is also stored permanently inside the SW-TPM non-volatile memory through `TPM2_EvictControl` command, since the key is flexible and can remain the same. Finally, the certificate and the AK are sent to the $\mathcal{O}$`rc`, which verifies that the signature is valid and that the creation of the key was correct; if the verification is successful, the *VF* is included in the service graph chain.

4.2.3 PCR management

While standard static PCRs cannot be reset during run-time, NV-based PCRs do not follow the same rule and can actually be deleted and recreated indefinitely depending on how they are created. It is therefore necessary to enforce some rules such that NV-PCRs behave in the same way as normal PCRs. Specifically, the NV-PCRs will be created with binding to a flexible policy so that only the $\mathcal{O}$`rc` may authorise operations on the NV indexes. In particular, only deletion requests by the orchestrator will be allowed to actually undefine the NV index; by including into the authorisation policy of the NV-PCRs the command code of `TPM2_NV_UndefineSpaceSpecial`, only specifically authorised policies will be able to undefine the NV index; finally, only the NV index specified can be undefined by a *VF* since the $\mathcal{O}$`rc` will also include a digest over the name of that index.

Figure 4.7 shows the necessary steps for the attachment process for both normal and NV-Based PCRs. In the case of normal PCRs the process is greatly simplified, since it is only necessary for the $\mathcal{O}$`rc` to specify which PCRs are to be used to the virtual function. The orchestrator adds them to its own mPCRs structure associated to the *VF*, while the latter adds them to its normal PCR structure. Differently, when NV-based PCRs are to be attached, the $\mathcal{O}$`rc` has to send more information

- the identifier for the NV-PCRs.
- a template which described the characteristics of the NV slots, such as the hashing algorithm to be used, the extension operation via `TPM2_NV_Extend` and that a policy is required to authorise the deletion of the index.
- the authorisation policy which exclusively authorises the orchestrator to request the index deletion, as described in the previous paragraph.
- an initial value (IV) to be extended by the newly created NV-PCRs, since the latter can only be certified when written at least once.

Once all of this information is received by the *VF*, it invokes its TPM to perform all

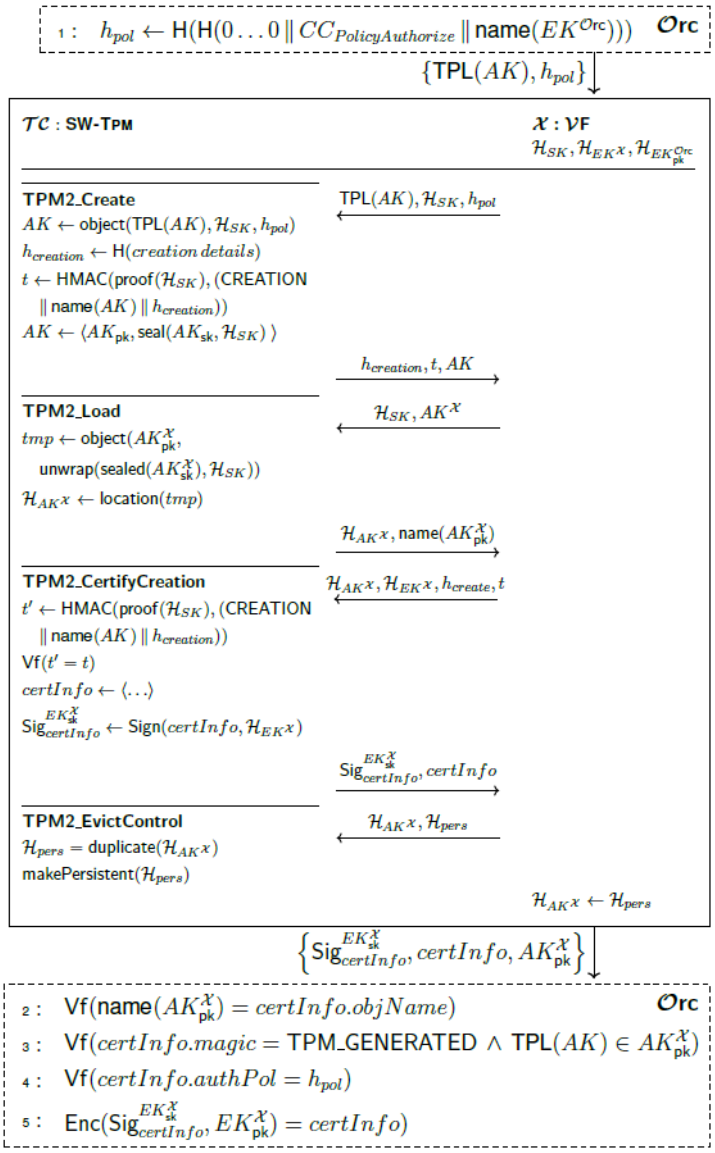

Figure 4.6. AK creation phase

the necessary operations for the creation, extension and certification of the newly requested NV-PCRs. The certification information is then sent back to the $\mathcal{O}\mathtt{rc}$, which in turn has to verify whether the certificate is authentic, that the structure was actually generated by the TPM and that it is associated to the correct value and authorisation policy previously specified.

When considering the detachment operation of PCRs, while with normal ones

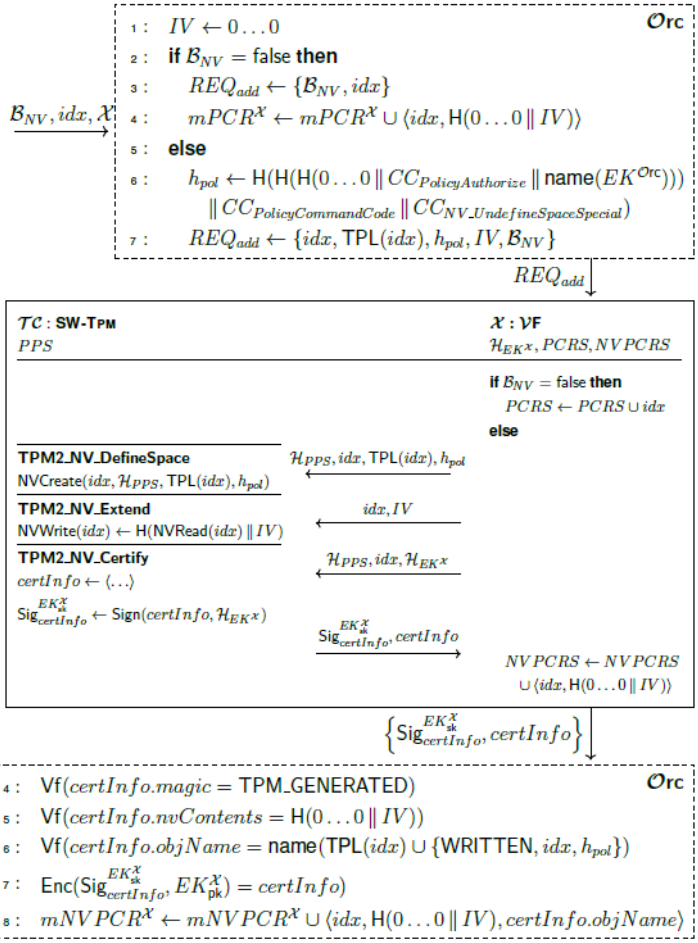

Figure 4.7. Attaching normal or NV-PCRs

the protocol is extremely simplified, with the $\mathcal{O}$rc just informing the *VF* about the PCRs to remove from its structure, the case of NV-PCRs is much more complex. Figure 4.12 shows all the steps required. The orchestrator requests the start of a new policy session to the *VF* and receives back the session-generated nonce. At this point, the $\mathcal{O}$rc locally executes *Algorithm 1*, reported in Figure 4.8. The purpose of the algorithm is to create a policy h_{pol}, such that the `TPM2_PolicySigned` can only be executed with a digest signed by the orchestrator, which is computed over the nonce received, an expiration and a further digest h_{cp} which is used to restrict the session to the `TPM2_UndefineSpaceSpecial` command, along with NV-PCRs identifier. At this point, the deletion request is passed on to the *VF*, which invokes its TPM to verify that h_{pol} was signed by the $\mathcal{O}$rc and to run all subsequent policy commands, which allow, if everything is actually correct, to set the policy session

in a state that certifies the fulfilment of the policy authorised by the orchestrator and thus grants the limited execution of `TPM2_NV_UndefineSpaceSpecial` which actually deletes the NV index from memory.

Algorithm 1: Authorizing NV index deletion

Input : $n, idx, \mathcal{H}_{EK}, mNVPCR$

Output: $\left\{ idx, h_{cp}, \mathsf{Sig}^{EK^{\mathcal{O}rc}_{sk}}_{aHash}, h_{pol}, \mathsf{H}(h_{pol}), \mathsf{Sig}^{EK^{\mathcal{O}rc}_{sk}}_{\mathsf{H}(h_{pol})} \right\}$

1 $h_{pol} \leftarrow \mathsf{H}(\mathsf{H}(0 \ldots 0 \,\|\, CC_{PolicySigned} \,\|\, \mathsf{name}(\mathcal{H}_{EK})))$

2 $\mathsf{Sig}^{EK^{\mathcal{O}rc}_{sk}}_{\mathsf{H}(h_{pol})} \leftarrow tpm.\mathsf{Sign}(\mathsf{H}(h_{pol}), \mathcal{H}_{EK}))$

3 $h_{cp} \leftarrow \emptyset$

4 $\forall \langle \mathcal{H}, h, \mathsf{name}(\mathcal{H}) \rangle \in mNVPCR$:

5 **if** $\mathcal{H} = idx$ **then**

6 $h_{cp} \leftarrow \mathsf{H}(CC_{NV\_UndefineSpaceSpecial} \,\|\, \mathsf{name}(\mathcal{H}) \,\|\, \mathcal{H}_{PPS})$

7 $aHash \leftarrow \mathsf{H}(n \,\|\, 0 \,\|\, h_{cp})$

8 $\mathsf{Sig}^{EK^{\mathcal{O}rc}_{sk}}_{aHash} \leftarrow tpm.\mathsf{Sign}(aHash, \mathcal{H}_{EK})$

9 **return** $idx, h_{cp}, \mathsf{Sig}^{EK^{\mathcal{O}rc}_{sk}}_{aHash}, h_{pol}, \mathsf{H}(h_{pol}), \mathsf{Sig}^{EK^{\mathcal{O}rc}_{sk}}_{\mathsf{H}(h_{pol})}$

Figure 4.8. Detailed steps of Algorithm 1

4.2.4 Measurement update

The measurement update phase is carried out with the same principle of the one described in Section 4.1.2. In fact, the orchestrator starts the request by locally performing the measurement process and updating the mPCR structure of the corresponding *VF* with the new values obtained by measuring the configuration pointed by a *Fully Qualified Path Name* (FQPN), which is also authenticated because the $\mathcal{O}$`rc` shares the same unique key with the *VF* attestation agent. The orchestrator then proceeds to apply *Algorithm 2*, shown in Figure 4.9, in order to compute the expected policy digest with the correct values of the mock PCRs. This digest, along with all information required to perform the measurement update, i.e. the FQPN, the PCR type and the index affected, is sent to the *VF*. Once the *VF* receives the data, the request is intercepted by its local $\mathcal{AA}$`gt`, which proceeds to obtain an authenticated measurement of the FQPN by using the $\mathcal{T}$`rce` utility. The *VF* then invokes the SW-TPM to verify that the received policy digest is actually signed by the $\mathcal{O}$`rc`, returning a ticket as proof in case of success. Afterwards, the PCR extension phase actually starts by using the PCR extend command in audit mode, in order to later provide to the orchestrator proof that the extension was performed correctly, by accumulating the CPs and their Response Parameters

Algorithm 2: Composing AK policy update requests

Input : $idx, \mathcal{B}_{NV}, h_{update}, mNVPCR, mPCR, \mathcal{H}_{EK}$

Output: $\left\{h_{pol}, \mathsf{H}(h_{pol}), \mathsf{Sig}^{\mathsf{k}}_{\mathsf{H}(h_{pol})}, idx, \mathcal{B}_{NV}\right\}$

1 **if** $\mathcal{B}_{NV} = \mathsf{true}$ **then**
2 $\quad \forall \langle \mathcal{H}, h, \mathsf{name}(\mathcal{H}) \rangle \in mNVPCR :$
3 $\quad\quad$ **if** $\mathcal{H} = idx$ **then** $h \leftarrow \mathsf{H}(h \,\|\, h_{update})$
4 **else**
5 $\quad \forall \langle idx', h \rangle \in mPCR :$
6 $\quad\quad$ **if** $idx' = idx$ **then** $h \leftarrow \mathsf{H}(h \,\|\, h_{update})$
7 **end**
8 $h_{\mathsf{pol}} \leftarrow 0 \ldots 0$
9 $\forall \langle \mathcal{H}, h, \mathsf{name}(\mathcal{H}) \rangle \in mNVPCR :$
10 $\quad args \leftarrow \mathsf{H}(h \,\|\, 0x0000 \,\|\, 0x0000)$
11 $\quad h_{pol} \leftarrow \mathsf{H}(h_{pol} \,\|\, CC_{PolicyNV} \,\|\, args \,\|\, \mathsf{name}(\mathcal{H}))$
12 **if** $mPCR \neq \emptyset$ **then**
13 $\quad h_{PCR} \leftarrow \emptyset, indices \leftarrow \emptyset$
14 $\quad \forall \langle idx', h \rangle \in mPCR :$
15 $\quad\quad h_{PCR} \leftarrow h_{PCR} \,\|\, h$
16 $\quad\quad indices \leftarrow indices \cup idx'$
17 $\quad h_{pol} \leftarrow \mathsf{H}(h_{pol} \| CC_{PolicyPCR} \| indices \| \mathsf{H}(h_{PCR}))$
18 **end**
19 $\mathsf{Sig}^{EK_{\mathsf{sk}}}_{\mathsf{H}(h_{pol})} \leftarrow tpm.\mathsf{Sign}(\mathsf{H}(h_{pol}), \mathcal{H}_{EK})$
20 **return** $h_{pol}, \mathsf{H}(h_{pol}), \mathsf{Sig}^{EK_{\mathsf{sk}}}_{\mathsf{H}(h_{pol})}, idx, \mathcal{B}_{NV}$

Figure 4.9. Detailed steps of Algorithm 2

Algorithm 3: Witness

Input : $\mathsf{CMD} : \mathcal{H}_0 \times \mathcal{H}_1 \times \mathcal{H}_2 \times params \rightarrow RC \times CC \times rparams$

Output: h'_{audit} - updated audit session digest

1 $cpHash \leftarrow \mathsf{H}(CC(\mathsf{CMD}) \,\|\, \mathsf{name}(\mathcal{H}_0) \,\|\, \mathsf{name}(\mathcal{H}_1) \,\|\, \mathsf{name}(\mathcal{H}_2) \,\|\, params)$
2 $rpHash \leftarrow \mathsf{H}(RC(\mathsf{Eval}(\mathsf{CMD})) \,\|\, CC_{\mathsf{CMD}} \,\|\, rparams)$
3 $h'_{audit} \leftarrow \mathsf{H}(h_{audit} \,\|\, cpHash \,\|\, rpHash)$
4 **return** h'_{audit}

Figure 4.10. Detailed steps of Algorithm 3

(RPs) into an *auditDigest* (this procedure is detailed by Algorithm 3 in Figure 4.10). Once the operation is complete, the SW-TPM certifies this digest with its own EK and all the required data is sent back to the $\mathcal{O}\mathtt{rc}$. The latter applies Algorithm 4, shown in Figure 4.11, in order to compute the expected audit digest by using the correct command codes and a success response parameter; if the digests match and if the signature is valid then it is assumed that the process was successful. The whole protocol is shown below in Figure 4.13.

Algorithm 4: Verify session audit digest

Input : $idx, \mathcal{B}_{NV}, h_{update}, mNVPCR, mPCR, \mathsf{Sig}^{EK_{\mathsf{sk}}}_{auditInfo}, auditInfo, EK_{\mathsf{pk}}$
Output: $\mathcal{B}$

1 **if** $\mathcal{B}_{NV} = \mathsf{true}$ **then**
2 $\forall \langle \mathcal{H}, h, \mathsf{name}(\mathcal{H}) \rangle \in mNVPCR :$
3 **if** $\mathcal{H} = idx$ **then**
4 $cpHash \leftarrow \mathsf{H}(CC_{NV\_Extend} \,\|\, \mathsf{name}(\mathcal{H}) \,\|\, \mathsf{name}(\mathcal{H}) \,\|\, \mathsf{len}(h_{update}) \,\|\, h_{update})$
5 $rpHash \leftarrow \mathsf{H}(\mathsf{success} \,\|\, CC_{NV\_Extend})$
6 **else**
7 $\forall \langle idx', h \rangle \in mPCR :$
8 **if** $idx' = idx$ **then**
9 $cpHash \leftarrow \mathsf{H}(CC_{PCR\_Extend} \,\|\, idx' \,\|\, idx' \,\|\, authHash \,\|\, h_{update})$
10 $rpHash \leftarrow \mathsf{H}(\mathsf{success} \,\|\, CC_{PCR\_Extend})$
11 **end**
12 $h_{audit} \leftarrow \mathsf{H}(0 \ldots 0 \,\|\, cpHash \,\|\, rpHash)$
13 $\mathsf{Vf}(h_{audit} = auditInfo.h_{session})$
14 $\mathsf{Vf}(\mathsf{Enc}(\mathsf{Sig}^{EK_{\mathsf{sk}}}_{auditInfo}, EK_{\mathsf{pk}}) = auditInfo)$
15 **return** true

Figure 4.11. Detailed steps of Algorithm 4

4.2.5 Oblivious remote attestation phase

The final step of the protocol is the one that actually allows the distributed verification of trust among nodes of the same service graph chain. Once the attestation keys have been created and distributed and each node is in a correct state with an authorised policy, it is possible to serve challenge requests between each other. When a *VF* receives a nonce n as a challenge to prove its conformance, it simply signs the value with its private AK and sends it back to the *Verifier*. The specific steps are reported in Figure 4.14. The virtual function that receives a challenge, first measures and verifies the content of all of its active PCRs (both normal and NV-based), then invokes `TPM2_PolicyAuthorize` to verify whether the current session policy digest matches the approved one and whether the ticket is valid. If the outcome is positive, then the current policy digest is replaced in the SW-TPM with the digest over the public part of the $\mathcal{O}$`rc`'s EK, which allows the use of the necessary attestation key for the digital signature.

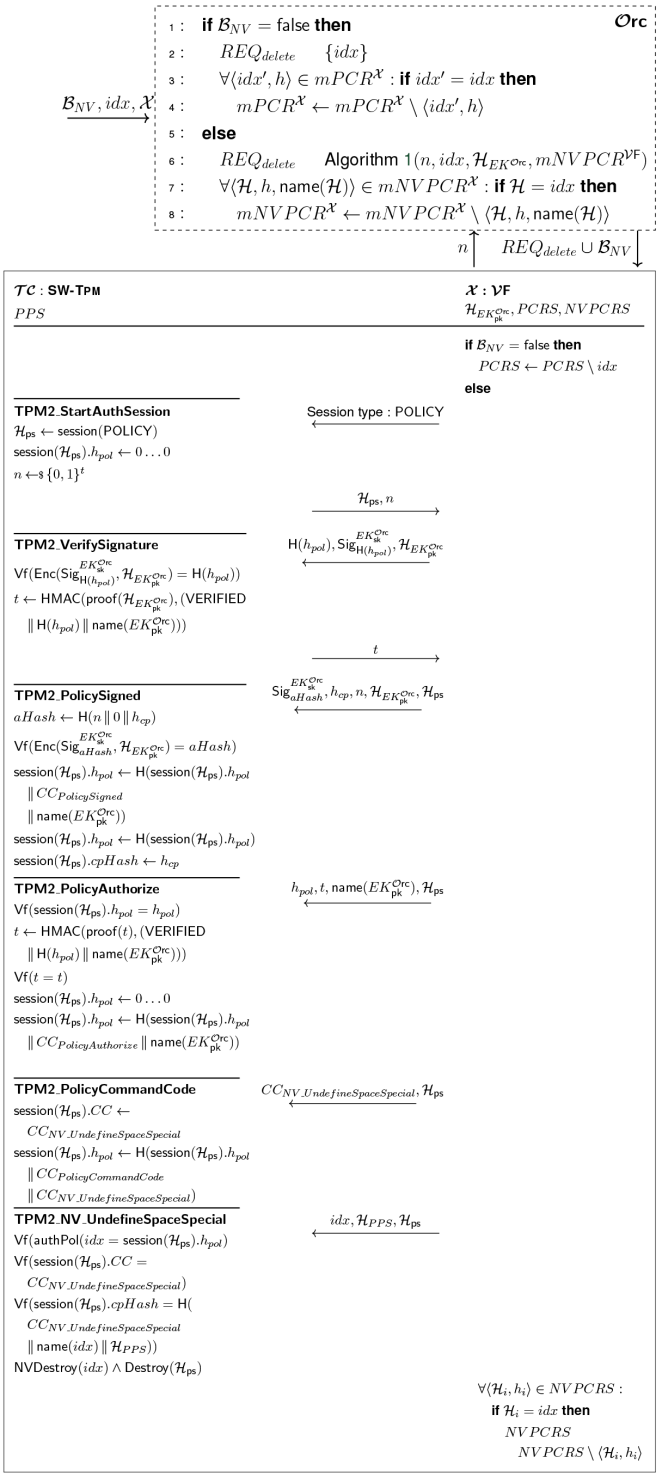

Figure 4.12. Detachment of normal or NV-PCRs

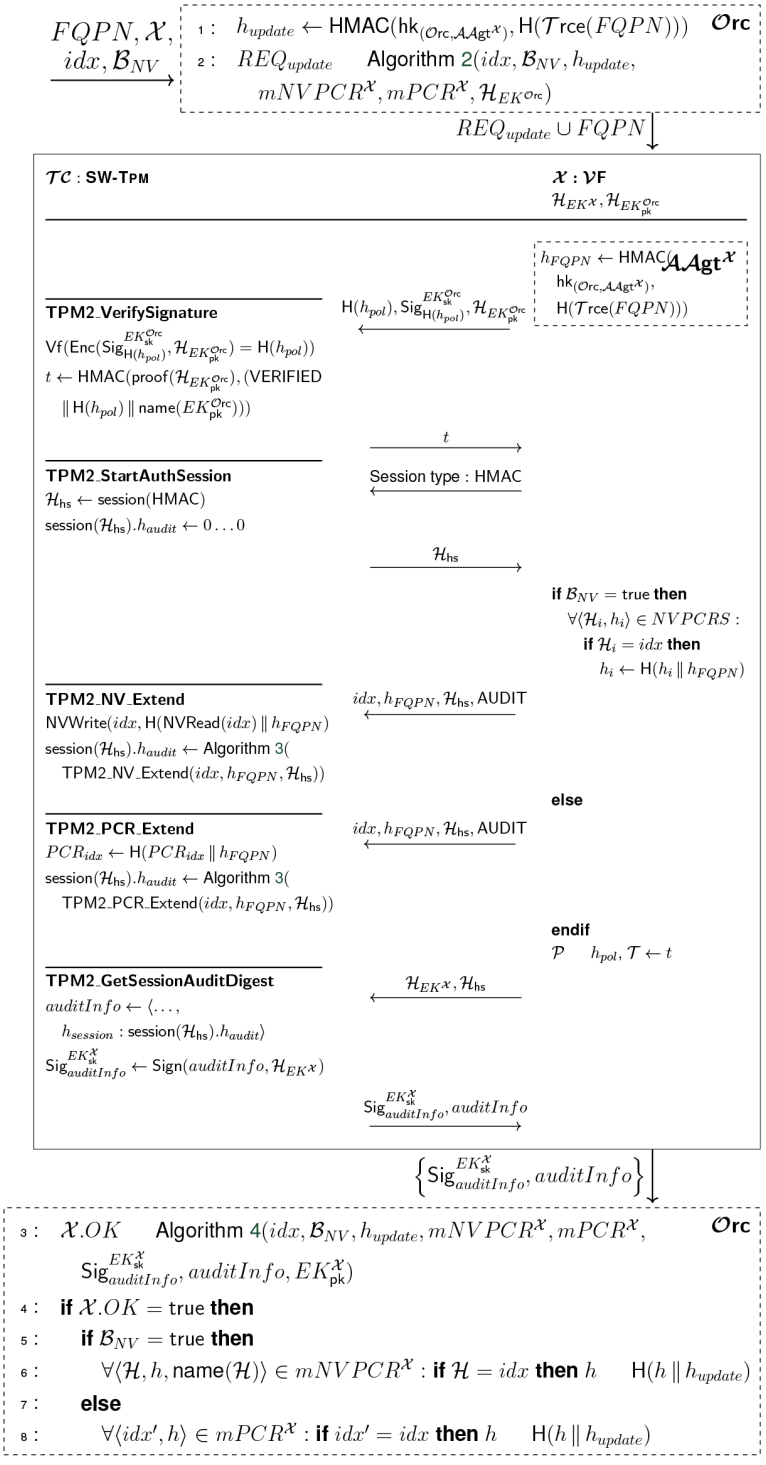

Figure 4.13. Measurement update phase

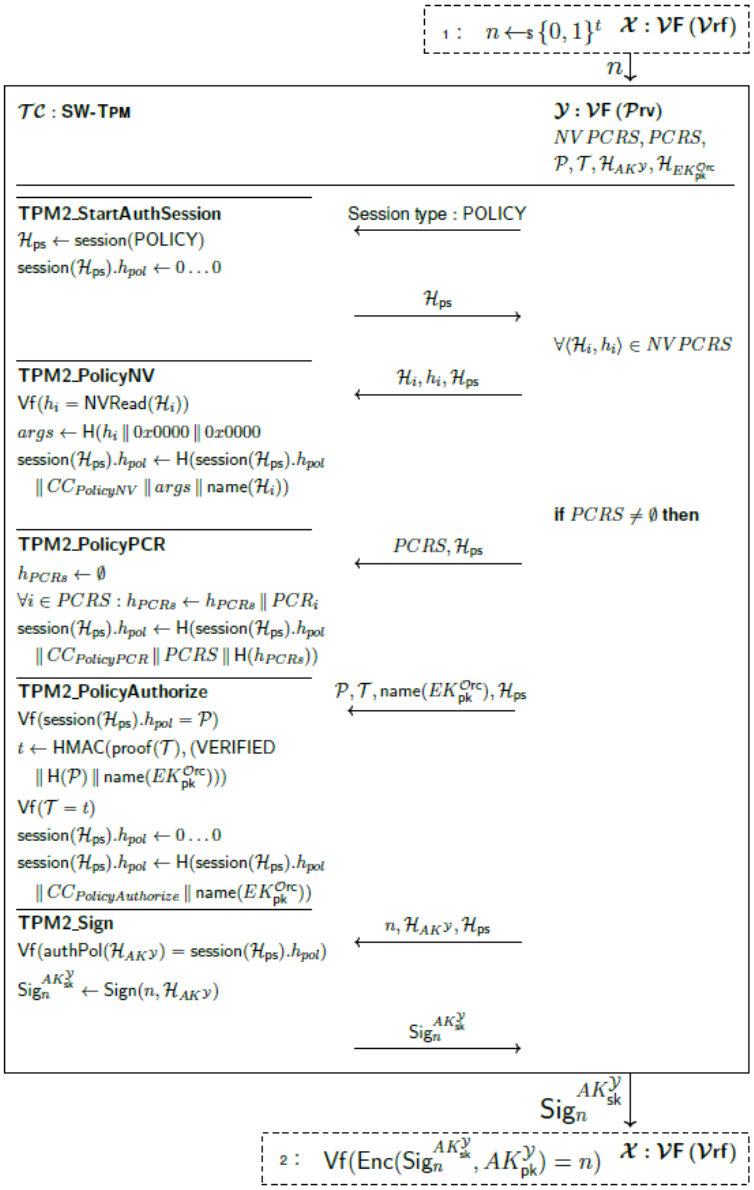

Figure 4.14. Oblivious remote attestation phase

Chapter 5

Thesis objective

The work carried out for this thesis is focused on the formal verification of protocols presented in Chapter 4 with the formal methods described in Chapter 2. In particular, in order to achieve the requirements of trust-aware service graph chains with zero-touch configuration functionalities, all while preserving the privacy of the participating entities, RAINBOW public Deliverable D2.1 [16] defines some specific properties

P1 Configuration Correctness. In order to maintain its trusted state, the configuration of a device must always adhere to the attestation policies currently deployed by the orchestrator, both at load-time and throughout run-time.

P2 SGC Trustworthiness. All nodes being part of a service chain must be in a trusted state and they must have provided to the orchestrator verifiable evidence about their correct behaviour. This follows as a consequence of P1, by assuming that all nodes have had their configuration integrity verified.

P3 Attestation Key Protection. The security of attestation keys, bound to specific authorisation policies, and in general of all keys involved in the protocols, must be guaranteed at all times. The secret part of the asymmetric keys must always be securely stored inside a TPM and never made available to an adversary.

P4 Immutability. The measurement process, given the same input data, must always return the same result, such that the tracer always gives the correct measurement.

One more property concerning the protection against timing attacks was specified, but as discussed in Chapter 2 these kind of side-channel attacks cannot be modelled with the formal verification methods employed in this work.

The following chapter will present the implementations of the protocols in the *ProVerif* specification language, as well as in some of its variants, with the aim of trying to verify the above-mentioned properties, through a combination of reachability queries and correspondence assertions, which model the desired results.

Chapter 6

Formal modelling of analysed protocols

The models presented in this chapter will only consist of the most relevant part, that is the actual processes interacting and the queries included, while mostly omitting the preamble with the definitions of types, functions and cryptographic primitives. A complete working copy of the models is included in the Appendix.

6.1 Attestation by quote

The modelling of this protocol proceeded incrementally with the attestation phase and the update request phase initially separated, testing various possible implementations in order to easily identify any possible shortcoming in its functionality.

6.1.1 Simple attestation phase

The first implementation consists of the standalone attestation by quote phase. In its simplest form, the protocol consists of an orchestrator entity which needs to verify whether the node entity is in a correct configuration state by checking specific PCR values. The main process starts by creating the asymmetric endorsement key EK belonging to the node of interest and then proceeds by outputting the public part into the public channel representing the network. The process also spawns two sub-processes, the `Orc` and the `TPM`, acting respectively as the orchestrator and the node to be verified. At this stage of development, the whole node is only represented by the its TPM, therefore excluding the chance of a dishonest device.

```
process
    new ek: ekpair;
    (!out(c, pk(ek));0 | !Orc(pk(ek)) | !TPM(sk(ek), pk(ek)))
```

Then the `Orc` process starts the interaction by producing a nonce n and the set of relevant indexes I and sending them to the `TPM` process. The latter retrieves the specified values from its PCRs through the utility function `extractRequestedPCR`, computes its hash `hConf` and constructs the quote structure with the hash, the nonce, and index set, while also producing a signature over it with its private EK. The implementation of the digital signature operation in the *ProVerif* environment allows to retrieve the message directly from the signature, therefore only the latter needs to be sent back to the orchestrator.

```
1  let Orc(pEK: publicEK) =
2      new n: bitstring;
3      new I: bitstring;
4      out(c, (n, I));
5      in(c, sig: bitstring);
6      let hConf = hash(extractRequestedPCR(I)) in
7      if checksign(sig, pEK) = ok() then
8          let (=hConf, =n, =I) = getmess(sig) in
9              event Trusted(); 0.
10
11 let TPM(sEK: secretEK, pEK: publicEK) =
12     in(c, (n: bitstring, I: bitstring));
13     let hConf = hash(extractRequestedPCR(I)) in
14         event SendingQuote();
15         out(c, sign((hConf, n, I), sEK));
16     0.
```

Once the orchestrator receives the signature it checks whether it is valid and then verifies that the quote structure matches the values expected. This check is expressed through the specific syntax of *ProVerif* pattern matching on line 8 above.

```
query event(Trusted()) ==> inj-event(SendingQuote()).
query event(Trusted()).
```

The queries implemented for this model involve the events embedded in the processes; in particular, one reachability query checks the execution of the `Trusted` event, which means that the signature and the values where all correct , whereas an injective correspondence query verifies that for each execution of the `Trusted`

event a unique execution of the `SendingQuote` event must have happened, ensuring that the potential trusted state is only relative to the last quote structure sent.

6.1.2 PCR update request and attestation

The second implementation combines the update and the attestation phases; most of the model is equal to the previous one with some key differences in the `Orc` and `TPM` processes and the queries used.

```
let Orc(pEK: publicEK) =
    new betaId: bitstring;
    new i: bitstring;
    new hBeta': bitstring;
    let vPCR = extendPCR(hash(hBeta')) in
        event UpdateTrustedConfiguration();
    out(c, (i, betaId));
    ...


let TPM(sEK: secretEK, pEK: publicEK) =
    in(c, (i: bitstring, betaId: bitstring));
    let beta = Tracer(betaId) in
        let hBeta = hash(beta) in
            let PCR = extendPCR(hash(hBeta)) in
                event UpdatedPCR();
    ...
```

At first, the orchestrator creates the identifier of the new configuration file(s), the hash of the latter and the set of the involved PCR indexes. After accumulating its vPCR values, the Orchestrator executes the event `UpdateTrustedConfiguration` and then sends the configuration identifier and the set of indexes on the public channel to the `TPM`, which upon receiving the data calls the `Tracer` function to measure the file, then computes the hash and extends the PCR content with the new values; at this point the event `UpdatedPCR` is executed.

Two additional correspondence queries are employed with this model and they are reported below

```
event(UpdatedPCR())==>inj-event(UpdateTrustedConfiguration()).
event(SendingQuote())==>event(UpdatedPCR()).
```

The first requires injective correspondence between two events, such that for each execution of `UpdatedPCR`, a unique execution of `UpdateTrustedConfiguration` must have happened; this verifies whether the update request is actually coming

from the orchestrator. The second query is a simple correspondence which ensures that the attestation phase only executes after the update phase.

6.1.3 Multiple fog nodes involvement

A third implementation was developed with the purpose of testing whether a specific fog node could be safely declared as trusted, when other nodes are concurrently interacting with the orchestrator. In particular, once again for the sake of simplicity the update phase was omitted, since it would not be necessary for this kind of test. Another sub-process is created by the main one at the start

```
process
    new targetek: ekpair;
    ( !Orc(pk(targetek)) |
    !Target_FogNodeTPM(pk(targetek), sk(targetek)) |
    !Other_FogNodeTPM())
```

While the node to verify receives its copy of the endorsement key, the other fog node will produce locally its EK and then perform the same operation carried out by the node of interest, i.e. create the Quote structure, signing it and sending it to the orchestrator.

```
let Other_FogNodeTPM =
    new ek: ekpair;
    out(c, pk(ek));
    in(c, (n: bitstring, I: bitstring));
    let hConf = hash(extractRequestedPCR(I)) in
        event SendingQuote(pk(ek));
        out(c, sign((hConf, n, I), sk(ek)));
    0.
```

The orchestrator in this case will receive an unknown public endorsement key and that will be the key used to perform the check on the signature it gets at the end. The orchestrator, however, also knows the actual EK belonging to the node it has to verify, so only if the key received and the actual one match, the node is declared as trusted.

```
let Orc(TpEK: publicEK) =
    in(c, XpEK: publicEK);
    new n: bitstring;
    new I: bitstring;
    out(c, (n, I));
    in(c, sig: bitstring);
```

```
    let hConf = hash(extractRequestedPCR(I)) in
    if checksign(sig, XpEK) = ok() then
        let (=hConf, =n, =I) = getmess(sig) in
        if (XpEK = TpEK) then event Trusted(TpEK);
    0.
```

The queries used this time almost the same as the ones in the first model, but in this case an additional constraint was included; the events are tied to a public endorsement key, so that, in particular, the correspondence query implies that the same key has to be used for both the creation of the Quote structure and the declaration of trust.

```
query key: publicEK;
event(Trusted(key)) ==> inj-event(SendingQuote(key)).
query key: publicEK;
event(Trusted(key)).
```

6.1.4 Fog node and TPM as separate entities

This next implementation increases the complexity of the model by separating the node and its TPM into two entities, which in this case are communicating through a public channel; the public channel allows the *Dolev-Yao* adversary to have complete control over the communication, so that it simulates a possibly vulnerable and/or compromised node. The main process of the model, apart from creating the endorsement key as usual, also creates the initial set of PCR values to pass both to the orchestrator and the TPM, in order to model the fact that the node was in a correct state before the new update phase.

```
process
    new targetek: ekpair;
    new oldPCR: PCRvalue;
    ( ORC(pk(targetek), oldPCR) | FogNode |
    TPM(pk(targetek), sk(targetek), oldPCR))
```

The execution of the sub-processes is similar to the previous one, with the exception of the `FogNode` which acts as a mediator between the orchestrator and the TPM, by passing the parameters and calling the tracer function to obtain binary data of the configuration.

```
let FogNode =
    in(c, (i:bitstring, betaID: bitstring));
    event PCR_UpdateRequestReceived();
    let hBeta = hash(Tracer(betaID)) in
        event PCR_Extend();
```

```
        out(node_tmp_c, (i, hBeta));

    in(c, (n: bitstring, I: bitstring));
    out(node_tmp_c, (n, I));
    in(node_tmp_c, sig: bitstring);
    out(c, sig);
    0.
```

The extension operation of the PCR is also explicitly modelled this time, with the use of the concatenate function both in the orchestrator and the TPM.

```
let ORC(pEK: publicEK, oldvPCR: PCRvalue) =
    new betaID: bitstring;
    new i: bitstring;
    event PCR_UpdateRequest();
    out(c, (i, betaID));

    new n: bitstring;
    new I: bitstring;
    out(c, (n, I));
    in(c, sig: bitstring);
    let hBeta' = hash(Tracer(betaID)) in
        let vPCR = hashPCR(concatenate(oldvPCR, hBeta')) in
            let hConf = hashPCR(getRequestedPCR(vPCR, I)) in
            if checksign(sig, pEK) = ok() then
                let (=hConf, =n, =I) = getmess(sig) in
                event TrustedNode(pEK);
    0.

let TPM(pEK: publicEK, sEK: secretEK, oldPCR: PCRvalue) =
    in(node_tmp_c, (i: bitstring, hBeta: bitstring));
    let PCR = hashPCR(concatenate(oldPCR, hBeta)) in
        event PCR_Updated();

    in(node_tmp_c, (n: bitstring, I:bitstring));
    let hConf = hashPCR(getRequestedPCR(PCR, I)) in
        event SendingQuoteStructure(pEK);
        out(node_tmp_c, sign((hConf, n, I), sEK));
    0.
```

Some additional events and queries have been included to verify the correct execution order of the operations.

```
query event
(PCR_UpdateRequestReceived()) ==> inj-event(PCR_UpdateRequest()).
query event
(PCR_Updated()) ==> inj-event(PCR_Extend()).
query event (PCR_Updated()).
```

The first injective correspondence is used to verify that the node actually receives the update request from the orchestrator, whereas the second one to verify that the actual update of the PCRs on the TPM follows a unique request from its node. The last reachability query verifies that the update is actually executed.

6.1.5 Verification of RAINBOW security properties

This is the most complete and complex model for the *Attestation by quote* protocol, aiming to verify the RAINBOW security properties mentioned in chapter 5. The main process in this case creates the usual three actors, namely the orchestrator, the node and the TPM, along with an additional actor which represent the tracer utility. This separation serves the purpose of modelling a possible intervention of an adversary in the communication to and from the tracer itself, thus helping to prove the *Immutability* property.

```
let Tracer =
    in(trc, cid: configid);
    event TracerServingRequest();
    let configData = retrieveConfigData(cid) in
        out(trc, configData);
    0.
```

The rest of the model follows the usual execution order, with refined functions for the retrieval and extension of PCR values and with additional events that serve as a base for more specific correspondence queries.

```
1  let ORC(pek: pekey, pcrSet: pcrset) =
2
3      (* Policy Update *)
4      new cid: configid;
5      new I: index;
6      event StartingConfigurationUpdateOrc(cid, I);
7      let configData = retrieveConfigData(cid) in
8      let newpcrSet = Set(pcrSet, I,
9      hash((Get(pcrSet, I),hash(configData)))) in
10     out(chan, (cid, I));
11
12     (* Attestation by quote *)
13     new n: nonce;
14     out(chan, (n, I));
15     in(chan, sig: bitstring);
16     let hconf = hash(Get(newpcrSet, I)) in
17     if checksign(sig, xpek) = ok() then
18     let (hconf': bitstring, n': nonce, I': index) = getmess(sig)
19     in
20             if hconf'=hconf && n'=n && I'=I then
```

```
21                  event NodeTrusted(pek)
22                  else event NodeNotTrusted(pek);
23      0.
24
25  let FogNode(pek: pekey) =
26      (* Configuration Update *)
27      in(chan, (cid: configid, I:index));
28      event StartingConfigurationUpdateFogNode(cid, I);
29      (* Request to the Tracer *)
30      out(trc, cid);
31      in(trc, configData: bitstring);
32      let hb = hash(configData) in
33          out(pchan, (hb, I));
34
35      (* Attestation by quote *)
36      in(chan, (n': nonce, I': index));
37      out(pchan, (n', I'));
38      in(pchan, sig': bitstring);
39      out(chan, sig');
40      0.
41
42
43  let TPM(pek: pekey, sek: sekey, pcrSet: pcrset) =
44      (* Configuration Update *)
45      in(pchan, (hb: bitstring, I: index));
46      event StartingConfigurationUpdateTPM(I);
47      let newpcrSet = Set(pcrSet, I, hash((Get(pcrSet, I), hb))) in
48
49      (* Attestation by quote *)
50      in(pchan, (n': nonce, I': index));
51      let hconf' = hash(Get(newpcrSet, I')) in
52           out(pchan, sign((hconf', n', I'), sek));
53      0.
```

It is worth pointing out that the `ORC` doesn't employ the tracer utility when retrieving the new configuration on line 7 and that is to model the fact that the orchestrator is inherently trusted and therefore its values must always be the correct ones. On lines 5 and 13 , the creation of the set of PCR indexes I and the nonce n is done with new specific types, respectively `index` and `nonce`, differently from the previous models, where a simple `bitstring` was used; this is because when outputting the values together on a channel, the order was not always correctly maintained by *ProVerif* and thus some errors could ensue in the input operation in other processes.

The queries devised in this case include two reachability ones, to verify whether the events for the declaration of trust are actually executed, and two correspondence assertions;

```
query key: pekey; event(NodeTrusted(key)).
query key: pekey; event(NodeNotTrusted(key)).
query cid:configid, I:index;
    inj-event(StartingConfigurationUpdateTPM(I)) ==>
        (inj-event(StartingConfigurationUpdateFogNode(cid, I)) ==>
        inj-event(StartingConfigurationUpdateOrc(cid, I))).
query key: pekey; inj-event(NodeTrusted(key)) ==>
    inj-event(TracerServingRequest()).
```

The third query is actually a triple injective correspondence check, such that the whole configuration update process has to start from the orchestrator and proceed to the node and then to the TPM. The last query verifies that the declaration of trust follows an actual execution of the tracer utility, in order to ensure that the correct data was used.

6.2 Oblivious remote attestation

This enhanced version of the protocol improves the functionalities of the previous one, while inevitably increasing the complexity of the necessary operations to perform at each step. For this reason, the modelled protocol phases presented in the next sections will be considered only by themselves, not actually interacting with each other. This is due to the fact that *ProVerif* would struggle to handle such a complex system as a whole.

6.2.1 Attestation Key creation

The first phase modelled is the request to create a repurposable attestation key, bound to a flexible policy, when a *VF* wishes to be included in the service chains maintained by the orchestrator. The main process starts by creating the key material for the EK of the orchestrator, then proceeds to create both the *Storage Key* (SK), necessary to create the attestation key, and the EK for the TPM entity. In this case, the keys management is more precise, as each one is bound to a handle which includes the pair of asymmetric keys, a template that describes the characteristics of the key, a policy to which the key can be bound and possibly another handle to the parent key in the hierarchy. Each element of the tuple composing the key handle is retrievable through specific functions, defined as rewrite rules and publicly available, except for the one return the private part of the key; the latter is declared as private, and can only be accessed by honest actors of the process, therefore ensuring that keys are never compromised. This initial step is shown below.

```
process
        (* Orchestrator *)
        new orcEK:keymat;
        out(chan, pk(orcEK));

        (* TPM *)
        (* Create storage key SK *)
        new skpair:keymat;
        new sktemplate:bitstring;
        new skpolicy:bitstring;
        let tpmSKh = createHandle(sktemplate,
        nullhandle, skpolicy, pk(skpair), sk(skpair)) in
        out(chan, pk(skpair));
        (* Create Endorsement Key EK *)
        new ekpair:keymat;
        new ektemplate:bitstring;
        new ekpolicy:bitstring;
        let tpmEKh = createHandle(ektemplate,
        nullhandle, ekpolicy, pk(ekpair), sk(ekpair)) in
        out(chan, pk(skpair));

        (!TPM(tpmSKh, tpmEKh) |
        !fogNode(tpmEKh, tpmSKh) |
        !Orc(orcEK, getPK(tpmEKh)))
```

The orchestrator then is in charge of starting the interaction, by creating the authorisation policy and the template information for the creation of the AK. All command codes and response parameters for the TPM operation have been modelled as immutable free names.

```
let Orc(orcEK:keymat, ptpmEK:pkey) =
        (* Create the AK policy *)
        let hpol = hash(hash((zeros, CCpolicyauthorize,
        hash((pkeyname(pk(orcEK)),skeyname(sk(orcEK)))))))) in
        (* Create AK template *)
        new template:bitstring;
        out(chan, (hpol, template));
        ...
```

All the subsequent interactions mostly take place between the node and its TPM, following all the necessary steps for the correct creation of the AK.

```
let fogNode(ekh:keyhandle, skh:keyhandle) =
        (* Get the AK policy and template *)
```

```
        in(chan, (hpol:bitstring, template:bitstring));

        (* Send TPM_create request *)
        out(tpmchan, (template, skh, hpol));
        (* Get TPM created AK *)
        in(tpmchan, (template':bitstring,
        skh':keyhandle, hpol':bitstring,
        pAK':pkey, sealedsAK':skey,
        hcreation':bitstring, t':bitstring));

        (* Send TPM_Load request *)
        out(tpmchan, (template', skh', hpol', pAK', sealedsAK'));
        in(tpmchan, (akh:keyhandle, pakname:bitstring));

        (* Send TPM_certifyCreation request *)
        out(tpmchan, (akh, ekh, hcreation', t'));
        in(tpmchan, (certinfo:bitstring, signature:bitstring));

        (* Send all the data to the orc *)
        out(chan, (certinfo, signature, getPK(akh)));

        0.

let TPM(tpmSKh:keyhandle, tpmEKh:keyhandle) =
        (* TPM2_Create *)
        in(tpmchan, (template:bitstring,
        skh:keyhandle, hpol:bitstring));
        if skh = tpmSKh then
        (* Generate the new key *)
        new ak:keymat;
        new hcreation:bitstring;
        let t = hmac(tpmproof,
        (CREATION,
        (pkeyname(pk(ak)), skeyname(sk(ak))), hcreation)) in

        (* Seal AK secret part with sSK *)
        let sealedsAK = seal(sk(ak), getPK(tpmSKh)) in
        let pAK = pk(ak) in
        (* Create the wrap *)
        let wrap = (template, skh, hpol,
        pAK, sealedsAK, hcreation, t) in
        out(tpmchan, wrap);

        (* TPM2_Load *)
        in(tpmchan, (template':bitstring, skh':keyhandle,
        hpol':bitstring, pAK':pkey, sealedsAK':skey));
        if skh' = tpmSKh then
        let sAK' = unseal(sealedsAK', getSK(tpmSKh)) in
        let akh = createHandle(template', skh',
```

```
        hpol', pAK', sAK') in
        out(tpmchan, (akh, pkeyname(pAK')));

        (* TPM2_CertifyCreation *)
        in(tpmchan, (akh'':keyhandle,
        ekh'':keyhandle, hcreation'':bitstring,
        t'':bitstring));
        (* Get the AK key from the handler *)
        let pak'' = getPK(akh'') in
        let sak'' = getSK(akh'') in
        let xt = hmac(tpmproof,
        (CREATION, (pkeyname(pak''),
        skeyname(sak'')), hcreation'')) in
        if xt = t'' then        (* The key is associated to
        the right ticket and has been created
        inside the TPM *)
        let certinfo = (pkeyname(pak''),
        TPM_GENERATED, getTemplate(akh''),
        getPolicy(akh'')) in
        if ekh'' = tpmEKh then
        out(tpmchan, (certinfo, sign(certinfo, getSK(ekh'')))));
        0.
```

Once the key has been certified, both the certificate and the public part of the key are sent back to the orchestrator, which in turn verifies that all parameters match the expected ones. In case the check is successful and the AK is correctly created, the orchestrator sends a secret value into the public channel, whose reachability is then evaluated through a query that allows to know whether the protocol actually terminated.

```
let Orc(orcEK:keymat, ptpmEK:pkey) =
    ...
        (* Get the AK public part,
        the certificate and the signature
        over the certificate *)
        in(chan, (certinfo:bitstring,
        signature:bitstring, pak:pkey));
        if checksign(signature, ptpmEK) = ok() then
        let (objname:bitstring,
        magic:bitstring,
        magic':bitstring, authpol:bitstring) = certinfo in
        if objname = pkeyname(pak) then
        if magic = TPM_GENERATED then
        if magic' = template then
        if authpol = hpol then
        (* AK matches the policies *)
        out(chan, s1);
        0.
```

6.2.2 PCR or NV-PCR attachment

This phase describes the procedure necessary to attach new PCRs, that is add new values to the PCR structure of both the VF and the $\mathcal{O}$rc. This procedure is valid for both normal and NV-based PCRs, however, this model will only be considering the case of NV-PCRs, since, as described in Chapter 4 the attachment of normal PCRs is fairly straightforward, whereas NV-PCRs need to have a specific policy associated to their creation, in order to avoid unauthorised deletions not coming from the orchestrator.

At the start of the main process the EKs for both the Orc and the TPM are created, with the same handle tuple as the previous case, then all three sub-process entities are spawned.

```
process
        (* Orchestrator EK *)
        new orcekpair:keymat;
        let orcEKh = createHandle(nulltemplate, nullhandle,
        nullpolicy, pk(orcekpair), sk(orcekpair)) in
        out(chan, pk(orcekpair));

        (* TPM EK *)
        new tpmekpair:keymat;
        let tpmEKh = createHandle(nulltemplate, nullhandle,
        nullpolicy, pk(tpmekpair), sk(tpmekpair)) in
        out(chan, pk(tpmekpair));

        ( TPM(tpmEKh) | Orc(orcEKh, pk(tpmekpair)) |
        fogNode(tpmEKh) )
```

The orchestrator starts the interaction by building the request for the creation of new NV-PCRs. The request includes the authorisation policy, in which the command codes are defined as free names in the preamble of the program, the template, the index identifiers and the initial value (IV) that needs to be extended into the newly created NV-PCRs in order to certify them; in the model the IV is set to zero.

```
let Orc(orcEKh:keyhandle, ptpmEK:pkey) =
        (* Build request to attach a new nv-pcr *)

        let hpol = hash((hash(hash((zeros, CCpolicyauthorize,
        (pkeyname(getPK(orcEKh)), skeyname(getSK(orcEKh))))))),
            CCpolicycommandcode,
            CCnvundefinespacespecial)) in
        new tpl:bitstring;
        new idx:bitstring;
        let IV = zeros in
        out(chan, (idx, tpl, hpol, IV)); ...
```

The `fogNode` process does not execute any particular operation except for the exchange of messages and TPM invocation, therefore its code is omitted. The `TPM` process sequentially executes the phases of NV-PCR creation, extension and certification. In particular, the PCR structure is created in a similar manner as the keys, i.e. a handle that refers to a tuple of data including the index identifiers, the template, the authorisation policy and most importantly a private channel (locally created in the process session) that models the actual content of the PCRs, following the flawed representation of state in the standard *ProVerif* language.

```
let TPM(tpmEKh:keyhandle) =
        (* TPM_NVDefineSpace *)
        in(tpmchan, (idx:bitstring, tpl:bitstring,
            hpol:bitstring));
        new nvpcrcontchan:channel;
        (* nv content initialized to zeros *)
        out(nvpcrcontchan, zeros);
        let nvprc = createNVPCR(idx, nvpcrcontchan, hpol, tpl) in
        out(tpmchan, nvprc);

        (* TPM_NVExtend *)
        in(tpmchan, (nvpcr':nvpcrhandle, IV':bitstring));
        let nvpcrcontchan' = getPCRContentChannel(nvpcr') in
        (* Extend the value inside the PCR *)
        in(nvpcrcontchan', oldvalue:bitstring);
        out(nvpcrcontchan', hash((oldvalue, IV')));

        (* TPM_NVCertify *)
        in(tpmchan, (nvpcr'':nvpcrhandle, ekh:keyhandle));
        if ekh = tpmEKh then
        let sEK = getSK(ekh) in
        let tpl'' = getPCRTemplate(nvpcr'') in
        let hpol'' = getPCRPolicy(nvpcr'') in
        let idx'' = getPCRIndex(nvpcr'') in
        let nvpcrcontchan'' = getPCRContentChannel(nvpcr'') in
        in(nvpcrcontchan'', content'':bitstring);
        let certInfo = (TPM_GENERATED, content'',
        hash((tpl'', WRITTEN, idx'', hpol''))) in
        out(tpmchan, (certInfo, sign(certInfo, sEK)));
```

Once the new NV-PCRs are extended and certified, the certification info is sent back to the orchestrator that, as usual, verifies that everything was executed correctly by checking the signature over the certificate and the content specifying that the creation actually happened inside the SW-TPM and that the content was extended as expected. In case of success, the NV-PCRs are attached to the structure.

```
let Orc(orcEKh:keyhandle, ptpmEK:pkey) =
    ...
    in(chan, (certInfo:bitstring, signature:bitstring));
        if checksign(signature, ptpmEK) = ok() then
        let (magic:bitstring,
        nvcontent:bitstring,
        objname:bitstring) = certInfo in
        if magic = TPM_GENERATED then
        if nvcontent = hash((zeros, IV)) then
        if objname = hash((tpl, WRITTEN, idx, hpol)) then
        (* The created NV-PCR matches what has been required *)
        out(chan, s1);
```

6.2.3 PCR or NV-PCR detachment

Similarly to the previous phase, the detachment operation can be carried out for both normal and NV-based PCRs. The first case with normal PCRs is once again fairly simple, so the model proposed below will only consider the NV-PCRs detachment. This process is quite complex as it is necessary to verify that several constraints are respected, in order to actually delete the NV index from memory. The main *ProVerif* process this time, apart from the usual creation of the EK for both the orchestrator and the TPM, also creates the initial existing NV-PCR structure that needs to be deleted and assigns to it the policy that will have to be matched in order to authorise the deletion.

```
process
        ...
        (* Create nv register *)
        new template:bitstring;
        let policy = hash((hash(hash((zeros, CCpolicyauthorize,
        (pkeyname(getPK(orcEKh)), skeyname(getSK(orcEKh)))))),
                 CCpolicycommandcode,
                 CCnvundefinespacespecial)) in
        new index:bitstring;
        new nvcontent:channel;
        let nvpcrh = createNVPCR(index, nvcontent,
        policy, template) in ...
```

The orchestrator starts the procedure for the deletion by requesting the involved node to create a fresh policy session and to get the session generated nonce; in this model it is directly the `fogNode` process which invokes the TPM to start the new session, which in turn verifies that the command was correct and generates the nonce n to pass back to the orchestrator.

```
let fogNode(tpmEKh:keyhandle, orcEKh:keyhandle) =
        (* Start policy session request *)
```

```
        out(tpmchan, START_POLICY_SESSION);
        in(tpmchan, nonce:bitstring);
        out(chan, nonce); ...

let TPM(tpmEKh:keyhandle, orcEKh:keyhandle) =
        (* Start policy session *)
        in(tpmchan, command:bitstring);
        if command = START_POLICY_SESSION then
        new nonce:bitstring;
        new sessionhpol:channel;
        new sessioncphash:channel;
        out(sessionhpol, zeros);
        out(tpmchan, nonce);
```

Once the orchestrator has received the nonce, it executes *Algorithm 1* (Figure 4.8) creating the `hpol` policy, its signature, as well as the digest `ahash` and its signature. These are needed to ensure that the `PolicySigned` command is executed with a policy authorised by the orchestartor and that the session is then restricted to the use of the `UndefineSpaceSpecial` command.

```
let Orc(orcEKh:keyhandle, ptpmEK:pkey, nvpcrh:nvpcrhandle) =
        (* Algorithm 1: auth nv index deletion *)
        in(chan, nonce:bitstring);
        let idx = getPCRIndex(nvpcrh) in
        let hpol = hash(hash(
        (zeros, CCpolicysigned, pkeyname(getPK(orcEKh)))))) in
        let hhpol = hash(hpol) in
        let signhhpol = sign(hhpol, getSK(orcEKh)) in
        let hcp = hash((CCnvundefinespacespecial, idx)) in
        let ahash = hash((nonce, zeros, hcp)) in
        let signahash = sign(ahash, getSK(orcEKh)) in

        out(chan, (nvpcrh, hcp, signahash,
        hpol, hhpol, signhhpol));
```

The rest of the code for the `fogNode` process is omitted as it only consists of a sequence of commands to the TPM, whereas the latter process carries out all the necessary steps by accumulating the policy digests computed in two local channels, used as state variables.

```
...
(* TPM_VerifySignature *)
        in(tpmchan, (hhpol:bitstring,
        signhhpol:bitstring, handle:keyhandle));
        if handle = orcEKh then
        if checksign(signhhpol, getPK(orcEKh)) = ok then
```

```
if getmess(signhhpol) = hhpol then
let t = hmac(tpmproof,
(VERIFIED, hhpol, pkeyname(getPK(orcEKh)))) in
out(tpmchan, t);

(* TPM_PolicySigned *)
in(tpmchan, (signahash':bitstring,
hcp':bitstring, nonce':bitstring, handle':keyhandle));
if handle' = orcEKh then
let ahash = hash((nonce', zeros, hcp')) in
if checksign(signahash', getPK(orcEKh)) = ok then
if getmess(signahash') = ahash then
in(sessionhpol, oldsessionhpol':bitstring);
out(sessionhpol, hash(hash(
(oldsessionhpol', CCpolicysigned,
pkeyname(getPK(orcEKh))))));
out(sessioncphash, hcp');

(* TPM_PolicyAuthorize *)
in(tpmchan, (hpol'':bitstring,
t'':bitstring, orcname'':bitstring));
in(sessionhpol, oldsessionhpol'':bitstring);
if oldsessionhpol'' = hpol'' then
let newt = hmac(tpmproof,
(VERIFIED, hash(hpol''), orcname'')) in
if newt = t'' then
out(sessionhpol,
hash((zeros, CCpolicyauthorize, orcname'')));

(* TPM_PolicyCommandCode *)
in(tpmchan, commandcode:bitstring);
let sessioncc = commandcode in
in(sessionhpol, oldsessionhpol''':bitstring);
out(sessionhpol, hash((oldsessionhpol''',
CCpolicycommandcode, CCnvundefinespacespecial)));

(* TPM_NVUndefineSpaceSpecial *)
in(tpmchan, nvpcrh:nvpcrhandle);
if sessioncc = CCpolicycommandcode then
in(sessionhpol, hpolvalue:bitstring);
if hpolvalue = getPCRPolicy(nvpcrh) then
in(sessioncphash, cphashvalue:bitstring);
if cphashvalue = hash((CCnvundefinespacespecial,
getPCRIndex(nvpcrh))) then
(* Deletion can be done *)
out(chan, s1);
```

The process concludes at this point, as there is no need to return to the orchestrator for additional checks on the validity of the operations, since when the

session policy digest is in a state that marks the fulfilment of a policy authorised by the `Orc`, then the `UndefineSpaceSpecial` command can be directly executed, thus removing the specified NV index from memory. The value outputted at the end is used to verify reachability and the correct termination of the protocol.

6.2.4 Measurement update

This phase describes the procedure with which the orchestrator requests a specific node to update its current configuration by measuring the data pointed by the *Fully Qualified Path Name* and accumulate the new values into its PCRs. The main process in the model creates the EK for orchestrator and TPM and a PCR structure with an initial value which will have to be extended. The update may concern both normal and NV-based PCRs with minimal differences; the model presented below will take into account the update of normal PCRs.

```
process
        ...
    (* Create pcrs *)
    new index:bitstring;
    new pcrcontent:channel;
    out(pcrcontent, pcr_iv);
    let pcrh = createPCR(index,
    pcrcontent, nullpolicy,
    nulltemplate) in ...
```

The content of the PCR is once again modelled with a local channel used as state variable, following the typical implementation in standard *ProVerif*. The orchestrator starts by locally measuring a configuration obtained via the *FQPN* and authenticating the measurement result through its attestation agent that holds a unique shared with the *VF* of interest. It then executes *Algorithm 2* (Figure 4.9) to compute the expected policy digest by extending its mock PCR structure and then passes all necessary information to perform the update to the `fogNode` process.

```
let Orc(orcEKh:keyhandle, ptpmEK:pkey, pcrh:pcrhandle) =
    let hupdate = hmac(hk_orc_aagt, hash(tracer(fqpn))) in
    (* Algorithm 2 *)
    let pcrcontent = getPCRContentChannel(pcrh) in
    in(pcrcontent, oldvalue:bitstring);
    let newpcrvalue = hash((oldvalue, hupdate)) in
    out(pcrcontent, newpcrvalue);
    let hpol = hash((zeros, CCpolicypcr,
    getPCRIndex(pcrh), hash(newpcrvalue))) in
    let hhpol = hash(hpol) in
```

```
    let signhhpol = sign(hhpol, getSK(orcEKh)) in
    out(chan, (hpol, signhhpol));
    ...
```

The `fogNode` process receives the data and passes it one to its SW-TPM which verifies that the signature and the policy digest by the orchestrator are valid, by producing a ticket as proof.

```
let fogNode(ptpmEK:pkey, porcEK:pkey) =
    in(chan, (hpol:bitstring, signhhpol:bitstring));
    out(tpmchan, (hpol, signhhpol));
    ...

let TPM(tpmEKh:keyhandle, orcEKh:keyhandle, pcrh:pcrhandle) =

    in(tpmchan, (hpol:bitstring, signhhpol:bitstring));

    (* TPM2_VerifySignature *)

    if checksign(signhhpol, getPK(orcEKh)) = ok() then
        if hash(hpol) = getmess(signhhpol) then
            let t = hmac(tpmproof, (VERIFIED, hash(hpol),
            pkeyname(getPK(orcEKh)))) in
            out(tpmchan, t);
    ...
```

In order to prove to the orchestrator that the update process is performed correctly, the node starts a *HMAC session* and runs the `extend` command in audit mode, so that every command and response parameter is recorded into a digest that will later be verified by the orchestrator. Specifically, the node uses its own attestation agent to measure the *FQPN* and authenticate that measurement that is then passed to the `TPM` process which performs the extension operation of the PCRs and also executes *Algorithm 3* (Figure 4.10) to create the session audit digest.

```
let fogNode(ptpmEK:pkey, porcEK:pkey) =
    ...
    in(tpmchan, t:bitstring);
    new SessionType_HMAC:bitstring;
    out(tpmchan, SessionType_HMAC);

    in(tpmchan, Hhs:bitstring);
    new AUDIT:bitstring;
    let hfqpn = hmac(hk_orc_aagt, hash(tracer(fqpn))) in
        out(tpmchan, (hfqpn, AUDIT));

    in(tpmchan, sigAuditInfo:bitstring);
    out(chan, sigAuditInfo);
```

```
let TPM(tpmEKh:keyhandle, orcEKh:keyhandle, pcrh:pcrhandle) =
    ...
    (* TPM2_StartAuthSession *)

    in(tpmchan, SessionType_HMAC:bitstring);
    new Hhs: bitstring;
    out(tpmchan, Hhs);

    (* TPM2_PCR_Extend *)

    in(tpmchan, (hfqpn:bitstring, AUDIT:bitstring));

    let pcrcontent = getPCRContentChannel(pcrh) in
    in(pcrcontent, oldvalue:bitstring);
    let newpcrvalue = hash((oldvalue, hfqpn)) in
    out(pcrcontent, newpcrvalue);
    (* Algorithm 3 *)
    let cpHash = hash((CCpcrextend, getPCRIndex(pcrh),
    authHash, hfqpn)) in
    let rpHash = hash((success, CCpcrextend)) in
    let haudit = hash((zeros, cpHash, rpHash)) in
    (* TPM2_GetSessionAuditDigest *)
    let sigAuditInfo = sign(haudit, getSK(tpmEKh)) in
    out(tpmchan, sigAuditInfo);
```

At the end, the signed audit digest is sent back to the orchestrator that executes *Algorithm 4* (Figure 4.11), by computing the expected audit digest with the correct parameters and a success response code. If the digest computed by the `Orc` matches the one received from the node and the signature over it is valid, then measurement is considered done correctly. A reachability query checks the secrecy of the value made public at the end to verify whether the protocol terminates.

```
let Orc(orcEKh:keyhandle, ptpmEK:pkey, pcrh:pcrhandle) =
    ...
    (* Algorithm 4 - Verify *)

    in(chan, sigAuditInfo: bitstring);
    if (checksign(sigAuditInfo, ptpmEK) = ok()) then
    let cpHash = hash((CCpcrextend,
    getPCRIndex(pcrh), authHash, hupdate)) in
    let rpHash = hash((success, CCpcrextend)) in
    let haudit = hash((zeros, cpHash, rpHash)) in
    if(getmess(sigAuditInfo) = haudit) then
    out(chan, s1);
```

6.2.5 ORA

The *Oblivious Remote Attestation* phase is the actual core of the whole protocol and it consists of the verification of trust between two *Virtual Functions* that belong to the same service chain and know nothing about each other states, except for the public part of their respective authorised attestation key. There is no orchestrator represented in the model, only two *VFs* interact, in the role of *Prover* and *Verifier*, with the addition of the SW-TPM of the *Prover*.

The main process creates the EK of the orchestrator, which is available to the *Prover* and will be needed for the computation of the correct policy digest, even though the orchestrator itself is not participating in the process. Furthermore, the attestation key is created, along with both a set of normal PCRs (in the form of a simple `bitstring`) and a set of NV-PCRs with the same handle of the previous protocol phases and whose content is again represented by a private channel. At this point it is also necessary to compute the correct authorisation policy which will be bound to the attestation key through a handle, as well as the ticket proving that the AK was created correctly.

```
process
    (* Orchestrator EK *)
    new orcekpair:keymat;
    let orcEKh = createHandle(nulltemplate, nullhandle,
    nullpolicy, pk(orcekpair), sk(orcekpair)) in
    out(chan, pk(orcekpair));

    (* Y Attestation Key *)
    new akpair: keymat;

    (* Create PCRs *)
    new pcrcontent:bitstring;

    (* Create nv register *)
    new template:bitstring;
    new index:bitstring;
    new nvcontent:channel;
    new nvpcr:bitstring;
    out(nvcontent, nvpcr);
        let nvpcrh = createNVPCR(index, nvcontent,
        nullpolicy, template) in

    let args = (nvpcr, zeros) in
    let nvstep = hash((zeros, CCpolicynv,
    args, nvpcrhandlename(nvpcrh))) in
    let pcrstep = hash((nvstep, CCpolicypcr,
    pcrcontent, hash(pcrcontent))) in
    let P = pcrstep in
```

```
    let T = hmac(ticketproof, (VERIFIED, hash(P),
    pkeyname(getPK(orcEKh)))) in
    let authPol = hash((P, CCpolicyauthorize,
    pkeyname(getPK(orcEKh)))) in
    let AKh = createHandle(nulltemplate, nullhandle,
    authPol, pk(akpair), sk(akpair)) in

    ( !X(getPK(AKh)) |
    !Y(P, T, AKh, getPK(orcEKh)) |
    !TPM(pcrcontent, nvpcrh) )
```

Afterwards, the `X` process, acting as *Verifier*, generates a nonce `n` as a challenge, sends it to `Y` and awaits the signature over it to verify whether the other node is in trusted state.

```
let X(ypAK:pkey) =
    new n:bitstring;
    out(chan, n);
    in(chan, sig:bitstring);
    if(checksign(sig, ypAK) = ok()) then
        out(chan, s1);

    0.
```

The nonce is received by `Y` which, as a consequence, invokes its TPM by starting a new policy type session which returns a session handle; the latter holds both an identifier for the session type and a private channel which serves as a state variable, accumulating the policy digest, initially set to zero. The *Prover* then goes on, interacting with the TPM for message exchange until the signature over the nonce is actually obtained.

```
let Y(P:bitstring, T:bitstring,
    AKh:keyhandle, porcEK:pkey) =
    in(chan, n:bitstring);

    out(tpmchan, START_POLICY_SESSION);
    in(tpmchan, Hps:sessionhandle);

    out(tpmchan, (P, T, pkeyname(porcEK)));

    out(tpmchan, (n, AKh));

    in(tpmchan, sig:bitstring);
    out(chan, sig);
```

```
let TPM(pcrcontent:bitstring, nvpcrh: nvpcrhandle) =
    (* TPM2_StartAuthSession *)
    in(tpmchan, policysession:bitstring);
    new hpolchan:channel;
    out(hpolchan, zeros);
    let Hps = createSession(policysession, hpolchan) in
    out(tpmchan, Hps);
    ...
```

The TPM proceeds to retrieve the values for both its normal and NV-based PCRs, and accumulates their hashed value, along with the correct policy command codes, into the policy digest, repeatedly reading and writing on the state variable channel.

```
let TPM(pcrcontent:bitstring, nvpcrh: nvpcrhandle) =
    ...
    (* TPM2_PolicyNV *)

    in(getPolicyDigest(Hps), emptyhpol:bitstring);
    in(getPCRContentChannel(nvpcrh), nvpcr:bitstring);
    let args = (nvpcr, zeros) in
    let nvhpol = hash((emptyhpol, CCpolicynv,
    args, nvpcrhandlename(nvpcrh))) in
    out(getPolicyDigest(Hps), nvhpol);

    (* TPM2_PolicyPCR *)

    in(getPolicyDigest(Hps), nvhpol':bitstring);
    let pcrhpol = hash((nvhpol', CCpolicypcr,
    pcrcontent, hash(pcrcontent))) in
    out(getPolicyDigest(Hps), pcrhpol);
    ...
```

Once all of the PCRs have been accounted for, the TPM can compute the proof ticket and check that the current policy digest matches the one authorised by the orchestrator and that the ticket matches the verified one. If everything holds, then the current policy digest is replaced with a new digest computed over the name of the orchestrator EK and the `PolicyAuthorize` command code. This allows the TPM to use the secret part of the AK to sign the nonce challenge, sending it back to the *Verifier*.

```
let TPM(pcrcontent:bitstring, nvpcrh: nvpcrhandle) =
```

```
...
(* TPM2_PolicyAuthorize *)

in(tpmchan,
(P:bitstring, T:bitstring, porcEKname:bitstring));
in(getPolicyDigest(Hps), hpol:bitstring);
let t = hmac(ticketproof,
(VERIFIED, hash(hpol), porcEKname)) in
if( (P = hpol) && (T = t) ) then
out(getPolicyDigest(Hps), zeros);
let newhpol =
hash((hpol, CCpolicyauthorize, porcEKname)) in
out(getPolicyDigest(Hps), newhpol);

in(tpmchan, (n:bitstring, AKh:keyhandle));
if(newhpol = getKeyPolicy(AKh)) then
    let sig = sign(n, getSK(AKh)) in
    out(tpmchan, sig);
```

At the end a reachability query checks whether the protocol can terminate correctly.

6.3 Variations with global state handling

The protocols described in the previous have also been modified to account for the management of global state. Instead of relying on the standard implementation of *ProVerif* with private channels, that implies the issues discussed in Section 2.1.1, two alternative state extensions were tested.

6.3.1 Attestation by quote

the first protocol analysed does not necessarily need the use of state variables to avoid verification error, but the different implementation was developed anyway to test its applicability.

Concerning the *StatVerif* [3] language extension, both the simple attestation protocol and the combination of update and attestation phases have been modified. The addition include the use of two type `cell` variables declared in the preamble of the file: one models the set of artificial PCRs on the orchestrator, and the other the set of actual PCRs on the node to verify.

```
cell orc_tpm: pcrset.
```

```
cell node_tpm: pcrset.
```

These variables are set with the same initial value in the main process of the model:

```
orc_tpm := pcrSet;
node_tpm := pcrSet;
```

Then their value is simply read, extended and set again, while also locking the access for concurrent sessions. Below is a minimal example on the orchestrator:

```
lock(orc_tpm);
read orc_tpm as pcrSet;

(* Policy Update *)
new cid: configid;
new I: index;
event StartingConfigurationUpdateOrc(cid, I);
let configData = retrieveConfigData(cid) in
let newpcrSet = Set(pcrSet, I,
hash((Get(pcrSet, I),hash(configData)))) in
out(chan, (cid, I));
orc_tpm := newpcrSet;
unlock(orc_tpm);
```

With the *GSVerif* [8] front-end extension instead, it was necessary to add two free names as private channels, marked with specific keyword `cell`. The two channels are then used in the same exact way as the cells of the *StatVerif* model:

```
free orc_tpm: channel [private, cell].
free node_tpm: channel [private, cell].
...

out(orc_tpm, currPCR);
out(node_tpm, currPCR);
...

in(orc_tpm, pcrSet: pcrset);

(* Policy Update *)
new cid: configid;
new I: index;
event StartingConfigurationUpdateOrc(cid, I);
let configData = retrieveConfigData(cid) in
```

```
let newpcrSet = Set(pcrSet, I,
hash((Get(pcrSet, I),hash(configData)))) in
out(orc_tpm, newpcrSet);
```

6.3.2 Oblivious remote attestation

This protocol is the one that should actually benefit from the use of global states for the repeated read and write operations necessary to handle the content of PCRs or policy digests.

StatVerif

The implementation with the *StatVerif* extension was limited to only two phases of the protocol, since it was determined that even with the simple update protocol of the previous section, the state handling was too complex and the time required for the analysis to terminate was not reasonable. In particular, global states as `cell` variables were introduced in the NV-PCRs attachment phase and in the ORA phase. The first one was modified by creating a cell for the content of the NV-PCR to be attached, which is then read and re-set multiple times during the process. For the attestation phase, instead, the cell was used to model the policy digest for the TPM session.

GSVerif

The implementations developed with the *GSVerif* extension were more thorough, as the results obtained with the *Attestation by quote* protocol were initially more promising.

All of the protocol phases were modified with the exception of the Attestation Key creation that needed no global state. The NV-PCRs attachment and detachment were modified by marking the private channels, used for PCR content and for policy digests, with the keyword `cell`. However, this implied some necessary modifications to the way the model handled its variables, as the GSVerif extension specifies that names marked as `cell` can only appear as the first argument of input and output operations. Therefore, the content channel had to removed from the tuple pointed by the NV-PCR handle. The code shows an example of this modification.

```
let TPM(tpmEKh:keyhandle) =
    ...
    new nvpcrcontchan:channel [cell];
        (* nv content initialized to zeros *)
```

```
        out(nvpcrcontchan, zeros);
        let nvprc = createNVPCR(idx, hpol, tpl) in
        out(tpmchan, nvprc);
        ...
        (* Extend the value inside the PCR *)
        in(nvpcrcontchan, oldvalue:bitstring);
        out(nvpcrcontchan, hash((oldvalue, IV')));
```

The measurement update phase was instead modified similarly to the *Attestation by quote* protocol, with two private channels modelling the content of both the orchestrator and TPM PCRs.

```
free pcrcontent_orc:channel [private, cell].
free pcrcontent_tpm:channel [private, cell].
...
(* In the main process *)
(* setting the same initial values *)
out(pcrcontent_orc, pcr_iv);
out(pcrcontent_tpm, pcr_iv);
...
(* In the Orc process *)
in(pcrcontent_orc, oldvalue:bitstring);
let newpcrvalue = hash((oldvalue, hupdate)) in
out(pcrcontent_orc, newpcrvalue);
```

Finally, the Oblivious Remote Attestation phase was modified with the introduction of the keywork `precise` for the channel that models the policy session digest. The keyword `precise`, which lets the *GSVerif* front-end automatically find the best option, was used in this case because the read and write operation on the channel inside an `if` construct did not allow the use of the keyword `cell` as in the other cases.

```
...
(* In the TPM *)
new hpolchan:channel [precise];
out(hpolchan, zeros);
(* TPM2_PolicyNV *)
in(hpolchan, emptyhpol:bitstring);
in(getPCRContentChannel(nvpcrh), nvpcr:bitstring);
let args = (nvpcr, zeros) in
let nvhpol = hash((emptyhpol, CCpolicynv,
args, nvpcrhandlename(nvpcrh))) in
out(hpolchan, nvhpol);
```

Chapter 7

Formal verification results

This chapter will report all the results obtained from the automatic analysis performed with the *ProVerif* tool and its extensions, relatively to the security properties specified by the RAINBOW platform and described in Chapter 5. First, the results for the *Attestation by quote* protocol will be presented, along with the possible attack traces identified. The results of the analysis on the ORA protocol will be reported after.

7.1 Attestation by quote

Considering the protocol versions implemented in Chapter 6 as a whole, three main issues were identified, with two of them producing an actual attack trace.

Unauthenticated Update Request. The update requests sent to the nodes to perform the measurement of the a new configuration and update its relative PCR values are actually not authenticated. For this reason, any adversary may impersonate the orchestrator and provide false data to the nodes, prompting them to perform the update procedure. This leads to a violation of the *Configuration Correctness* property, since nodes, by accumulating bogus values into their PCRs, will be in an incorrect configuration. Even when the update request is actually sent by the orchestrator, it could be intercepted by an attacker with *Dolev-Yao* capabilities and changed into arbitrary values. This can also lead to the incapacitation of the node, in the manner of a *DoS* attack, if an unchecked number of requests is received. The attack trace relative to this issue, automatically obtained by the *ProVerif* analysis, is shown in Figure 7.1. The enhanced version of the protocol solves the problem by having the orchestrator sign its requests with its *Endorsement Key*.

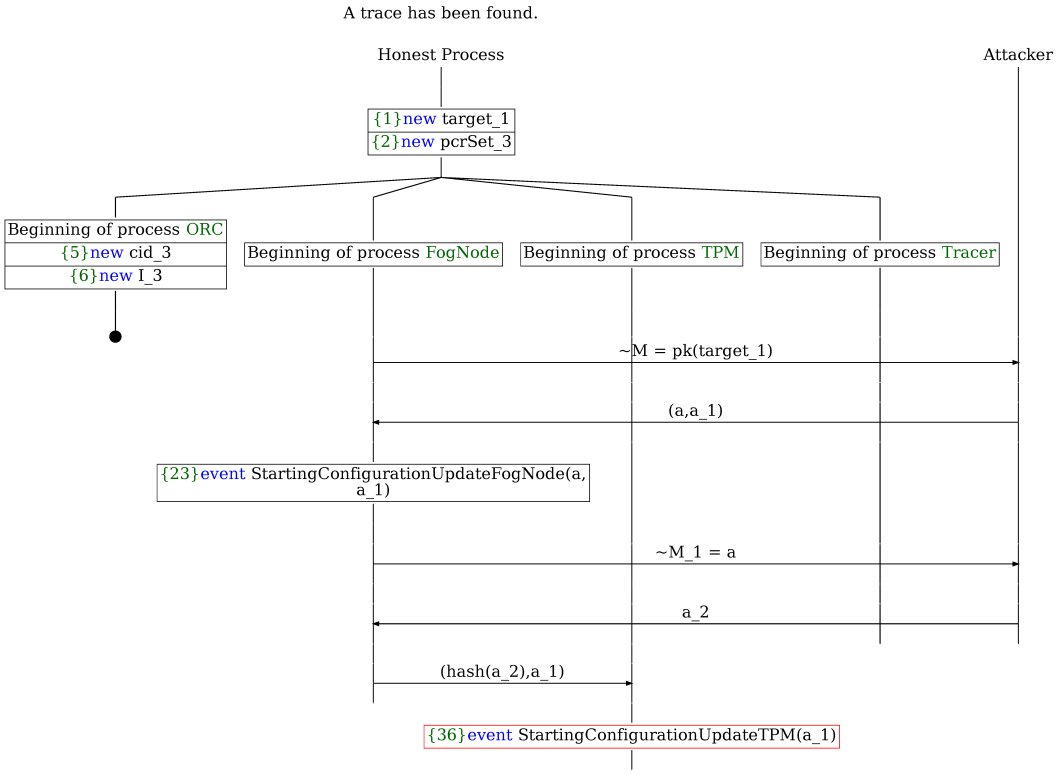

Figure 7.1. Attack trace showing the fake update request

Compromised Tracer Component. The measuring returned by the tracer is not authenticated and the communication between the component and the TPM that has to perform the update process may be exposed to an attacker, for instance in the case where the node equipped with the tracer has been compromised. The *Immutability* property is violated in this case, since the tracer could be returning the correct values that are modified with fake ones by an adversary. If an attacker has compromised a node, it could be able to produce the correct values by the tracer, such that the node would be considered trusted even though its configuration is not correct. This is shown in the attack trace in Figure 7.2. This issue as well is solved by the enhanced protocol, in particular by having each attestation agent authenticate the measurement with the unique key shared with the orchestrator.

Unauthenticated Attestation Request. This issue is basically a variation of the first one, but no trace was found by *ProVerif* relative to it, since the node configuration is not actually affected. If the attestation requests are not

signed by the orchestrator, any adversary may forge a request with bogus data and once again cause a DoS on the targeted node.

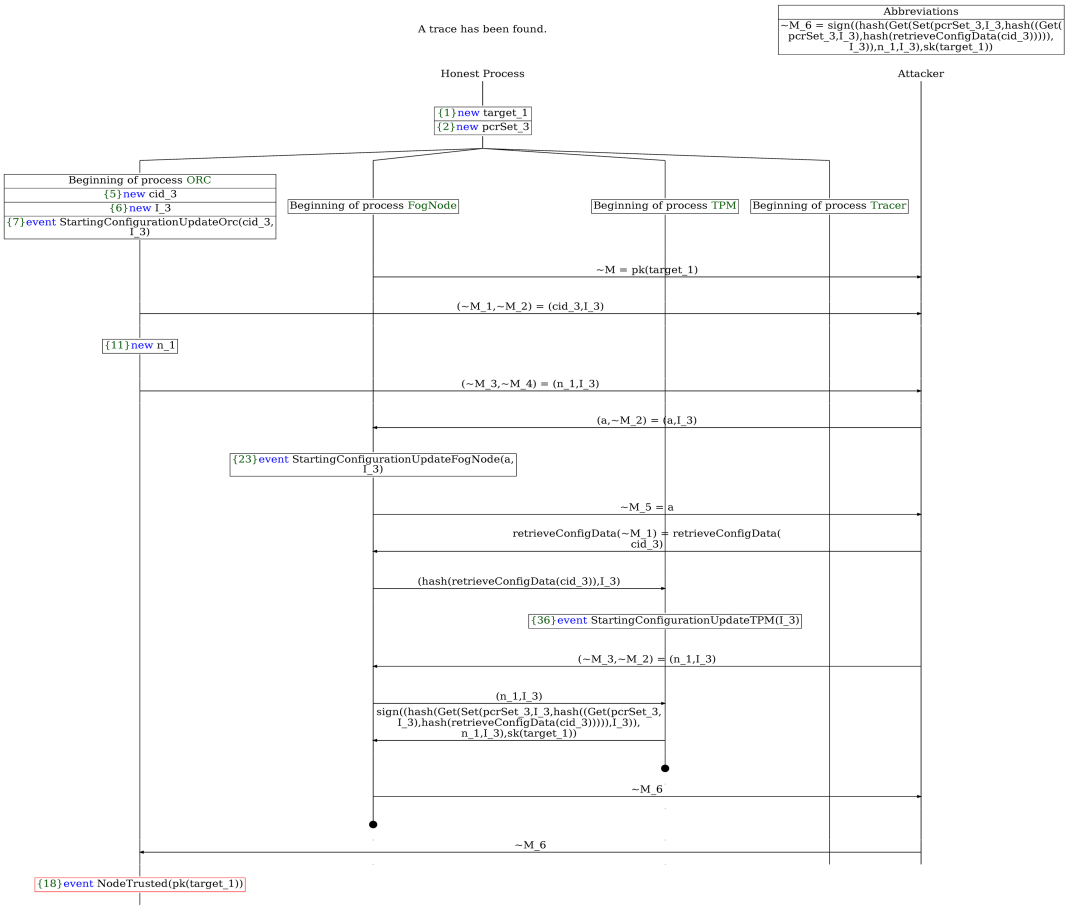

Figure 7.2. Attack trace with fake tracer values and node trusted

7.2 Oblivious remote attestation

The various phases of the protocol were analysed both with the standard *ProVerif* tool and with its global state extension. All the models included some reachability queries that aimed to assess whether the process could terminate correctly. Unfortunately, only the analysis on the model for the AK creation phase was able to reach a definitive proof and terminate with a positive result. In fact the creation of the attestation is performed correctly and the new key is securely stored into the SW-TPM of the *VF* that wishes to join the service chain.

For all the other phases *ProVerif* was not able to provide a definitive proof on

the validity of the queries requested. As a matter of fact, the derivations that *ProVerif* computes actually reach a termination point, but since no trace is found nothing can be inferred regarding the security of the system. This is most likely due to the high complexity of the protocol analysed, comprising of many different interactions between various actors. An example of such behaviour is reported below relatively to the Measurement Update phase of the protocol.

```
-- Query not attacker(s1[]) in process 1.
Translating the process into Horn clauses...
nounif mess(pcrcontent[],oldvalue_2)/-5000
Completing...
Starting query not attacker(s1[])
goal reachable: mess(pcrcontent[],oldvalue_2) && mess(pcrcontent
    [],oldvalue_3) && mess(pcrcontent[],oldvalue_4) -> attacker(s1
    [])
```

ProVerif translates the process into Horn clauses and then establishes a goal to reach to verify the requested query. The execution continues by reporting the whole derivation steps, as in:

```
1. We assume as hypothesis that
mess(pcrcontent[],oldvalue_2).

2. We assume as hypothesis that
mess(pcrcontent[],oldvalue_3).

...

9. The attacker has some term signhhpol_3.
attacker(signhhpol_3).

10. The attacker has some term hpol_3.
attacker(hpol_3).

11. By 10, the attacker may know hpol_3.
By 9, the attacker may know signhhpol_3.
Using the function 2-tuple the attacker may obtain (hpol_3,
    signhhpol_3).
attacker((hpol_3,signhhpol_3)).

...

19. The message oldvalue_2 that may be sent on channel pcrcontent
    [] by 1 may be received at input {34}.
The message sign(hash((zeros[],hash((CCpcrextend[],index[],
    authHash[],hmac(hk_orc_aagt[],hash(tracer(fqpn[])))))),hash((
    success[],CCpcrextend[])))),sk(tpmekpair[])) that the attacker
    may have by 18 may be received at input {41}.
So the message s1[] may be sent to the attacker at output {47}.
```

```
attacker(s1[]).

20. By 19, attacker(s1[]).
The goal is reached, represented in the following fact:
attacker(s1[]).
```

The last step concludes the derivation after the initially set goal is reached, however the tool simply could not find a trace which proves this derivation, therefore it cannot determine whether it is true or false.

```
Could not find a trace corresponding to this derivation.
RESULT not attacker(s1[]) cannot be proved.
```

Even when testing the global state extensions, which could supposedly improve the queries and therefore help the automatic analysis to terminate, no concrete results could be obtained. In particular, the analysis with *StatVerif* was completely unable to terminate since the multiple operations on the global state across concurrent sessions, quickly led to state explosion and an unreasonable computation time, as shown in the output below.

```
-- Query not attacker(s1[])
nounif attacker(cells(*hpol_186),v_187)/-5000
Completing...
200 rules inserted. The rule base contains 145 rules. 51 rules in
   the queue.
400 rules inserted. The rule base contains 214 rules. 115 rules in
    the queue.
600 rules inserted. The rule base contains 243 rules. 219 rules in
    the queue.
...
2800 rules inserted. The rule base contains 750 rules. 1307 rules
   in the queue.
3000 rules inserted. The rule base contains 809 rules. 1523 rules
   in the queue.
...
4200 rules inserted. The rule base contains 1038 rules. 2001 rules
    in the queue.
4400 rules inserted. The rule base contains 1089 rules. 2127 rules
    in the queue.
...
```

With the *GSVerif* translations, the analysis was able to terminate, but once again the results were the same as the standard *ProVerif* models, with no definitive proof on the queries. In particular, with these modifications, multiple reachable goals are analysed by the tool, which in the end are unified into a single clause that still contradicts the initial query. Since no derivation could be found, ProVerif arbitrarily terminates in order to avoid infinite loops.

```
Iterating unifyDerivation.
Fixpoint reached: nothing more to unify.
The clause after unifyDerivation is
begin(VCell_bitstring(pcrcontent_orc[],(0,pcr_iv[]))) && begin(
   VCell_bitstring(pcrcontent_orc[],(1,hash((pcr_iv[],hmac(
   hk_orc_aagt[],hash(tracer(fqpn[])))))))) && begin(
   VCell_bitstring(pcrcontent_tpm[],(0,pcr_iv[]))) && begin(
   VCell_bitstring(pcrcontent_tpm[],(1,hash((pcr_iv[],hmac(
   hk_orc_aagt[],hash(tracer(fqpn[])))))))) && begin(
   VCell_bitstring(pcrcontent_tpm[],(2,hash((hash((pcr_iv[],hmac(
   hk_orc_aagt[],hash(tracer(fqpn[]))))),hmac(hk_orc_aagt[],hash(
   tracer(fqpn[])))))))) -> attacker_bitstring(s1[])
This clause still contradicts the query.
Could not find a trace corresponding to this derivation.
Stopping attack reconstruction attempts.
```

7.3 Assessment on security properties

Concerning the RAINBOW that needed proving, some final evaluation can be given

P1 Configuration Correctness. The correct configuration of the device is guaranteed when the it performs the measurement update and the attestation to the orchestrator. This property is proved through an injective correspondence assertion in the *Attestation by quote* model.

P2 SGC Trustworthiness. All nodes participating in a service chain must be in a correct configuration state. This property follows automatically once the first one is proved; it can be assumed that if the protocol terminates correctly for a node without external interference, then all nodes in the chain should also be able to be safely declared as trusted. This is proved by a reachability query on the *Attestation by quote* model to ensure the process termination.

P3 Attestation Key Protection. Even though the ORA protocol could not be proved in its entirety, the AK creation phase was able to terminate correctly, therefore ensuring that the attestation keys are safely created and stored inside the TPM. This is proved through a reachability query in the *AK creation* model.

P4 Immutability. The data returned by the tracer must always be the correct one. This property is verified with authentication for the tracer and it is proved by a correspondence assertion.

Chapter 8

Conclusions

This chapter concludes the work done in this in thesis.

First of all, formal verification techniques were analysed for their specific application on cryptographic protocols. In particular, automated theorem proving tools such as *ProVerif* and its extensions were presented.

Furthermore, the trusted computing concept of remote attestation was analysed, along with its application on network protocols for a trusted computing environment, such as the one provided by the RAINBOW project.

The work then focused on the modelling in the formal specification of *ProVerif* of all the protocols analysed, with the objective to find an automatic proof for a certain number of security properties, which cannot intuitively be inferred.

The complete models were analysed with the automatic theorem proving tools described and some results were obtained. In particular, the security properties of interest could mostly be proved for the first protocol analysed, whereas the second one did not yield a satisfying outcome.

It was therefore proved that the *ProVerif* tool (as of version 2.02) was not the correct choice for the analysis of the protocols described, even when aided by its extensions aiming to improve its ability to terminate successfully on specific cases with states. The high complexity of the protocols is not suitable to be handled by the over-approximation applied by *ProVerif* when translating the model into *Horn clauses*.

Future work

This work easily leads to an extension, with the exploration of other automatic prover tools, such as *Tamarin*, which is natively able to deal with mutable states and allows direct manual interaction with the process when the automation fails.

Another possibility could be the use of an updated and over-hauled version of *ProVerif*, as described by its author [6], with improved precision and therefore ability to terminate more often, as well as optimisations in the internal algorithms that may improve its performance on complex cases.

Bibliography

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 36, 2001. ISSN 03621340. doi: 10.1145/373243.360213.

[2] M. Abadi, B. Blanchet, and C. Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *Journal of the ACM*, 65, 2017. ISSN 1557735X. doi: 10.1145/3127586.

[3] M. Arapinis, E. Ritter, and M. D. Ryan. Statverif: Verification of stateful processes. 2011. doi: 10.1109/CSF.2011.10.

[4] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 483–502, 2017. doi: 10.1109/SP.2017.26.

[5] B. Blanchet. Proverif: Cryptographic protocol verifier in the formal model. URL https://bblanche.gitlabpages.inria.fr/proverif/.

[6] B. Blanchet, V. Cheval, and V. Cortier. ProVerif with lemmas, induction, fast subsumption, and much more. In *IEEE Symposium on Security and Privacy (S&P'22)*, pages 205–222, San Francisco, CA, May 2022. IEEE Computer Society.

[7] L. Chen and M. Ryan. Attack, solution and verification for shared authorisation data in tcg tpm. volume 5983 LNCS, 2010. doi: 10.1007/978-3-642-12459-4_15.

[8] V. Cheval, V. Cortier, and M. Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. volume 2018-July, 2018. doi: 10.1109/CSF.2018.00032.

[9] P. B. Copet, G. Marchetto, R. Sisto, and L. Costa. Formal verification of LTE-UMTS and LTE–LTE handover procedures. *Computer Standards & Interfaces*, 50:92–106, 2017. ISSN 0920-5489. doi: https://doi.org/10.1016/j.csi.2016.08.009. URL https://www.sciencedirect.com/science/article/pii/S092054891630071X.

[10] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24, 1981. ISSN 15577317. doi: 10.1145/358722.358740.

[11] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29, 1983. ISSN 15579654. doi: 10.1109/TIT.1983.1056650.

[12] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. 2017. doi: 10.1109/EuroSP.2017.38.

[13] G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56, 1995. ISSN 00200190. doi: 10.1016/0020-0190(95)00144-2.

[14] The RAINBOW Consortium. Rainbow project h2020 - concept and objectives, 2020. URL https://rainbow-h2020.eu/concept-and-objectives/.

[15] The RAINBOW Consortium. Rainbow project h2020 - a fog computing platform, 2020. URL https://rainbow-h2020.eu/.

[16] The RAINBOW Consortium. D2.1, RAINBOW Attestation Model and Specification, 2021.

[17] The RAINBOW Consortium. D2.2, RAINBOW Collective Attestation Policy Enablers Design, 2021.

[18] The RAINBOW Consortium. D2.3, RAINBOW Collective Attestation and Runtime Verification, 2021.

[19] Trusted Computing Group. TCG Infrastructure Working Group Architecture Part II - Integrity Management, 2006.

[20] Trusted Computing Group. Trusted Platform Module Library Part 1: Architecture, 2019.

[21] Trusted Computing Group. Tpm 2.0 library specification, 2019. URL https://trustedcomputinggroup.org/resource/tpm-library-specification/.

Appendix A

Complete formal models implementation

A.1 Attestation By Quote

```
(* RAINBOW Attestation By Quote *)

set ignoreTypes = false.

type ekey.      (* endorsement key pair *)
type sekey.     (* secret endorsement key *)
type pekey.     (* public endorsement key *)
type pcrset.    (* complete set of pcr content *)
type pcr.       (* pcr content *)
type index.     (* pcr indexes *)
type configid.  (* configuration policy id *)
type nonce.

free chan:channel.  (* public network channel *)
 (* public channel to communicate with the tracer *)
free trc:channel.
(* private internal communication
between a node and its TPM *)
free pchan: channel [private].

(* Read configuration files *)
fun retrieveConfigData(configid): bitstring.

(* ----- cryptographic functions ----- *)
(* Cryptographic Hash *)
fun hash(bitstring): bitstring.
```

```
(* Digital Signature *)
type result.
fun sk(ekey): sekey.
fun pk(ekey): pekey.
fun ok(): result.
fun sign(bitstring, sekey): bitstring.
reduc forall m: bitstring, k: ekey;
getmess(sign(m, sk(k))) = m.
reduc forall m: bitstring, k: ekey;
checksign(sign(m, sk(k)), pk(k)) = ok().
(* ---------------------------------- *)

(* Get and Set *)
fun Get(pcrset, index):bitstring.
fun Set(pcrset, index, bitstring):pcrset.
equation forall pcrs:pcrset, i:index, c:bitstring;
Get(Set(pcrs,i,c),i) = c.

(* Protocol events *)
event StartingConfigurationUpdateOrc(configid, index).
event StartingConfigurationUpdateFogNode(configid, index).
event StartingConfigurationUpdateTPM(index).
event NodeTrusted(pekey).
event NodeNotTrusted(pekey).
event TracerServingRequest().

(* Queries *)
query key: pekey; event(NodeTrusted(key)).
query key: pekey; event(NodeNotTrusted(key)).
query cid:configid, I:index;
inj-event(StartingConfigurationUpdateTPM(I)) ==>
(inj-event(StartingConfigurationUpdateFogNode(cid, I)) ==>
inj-event(StartingConfigurationUpdateOrc(cid, I))).
query key: pekey;
inj-event(NodeTrusted(key)) ==>
inj-event(TracerServingRequest()).


let ORC(pek: pekey, pcrSet: pcrset) =

    (* Policy Update *)
    new cid: configid;
    new I: index;
    event StartingConfigurationUpdateOrc(cid, I);
    let configData = retrieveConfigData(cid) in
    let newpcrSet = Set(pcrSet, I,
    hash((Get(pcrSet, I),hash(configData)))) in
    out(chan, (cid, I));
```

```
    (* Attestation by quote *)
    new n: nonce;
    out(chan, (n, I));
    in(chan, sig: bitstring);
            let hconf = hash(Get(newpcrSet, I)) in
            if checksign(sig, pek) = ok() then
                let (hconf':bitstring,
                n':nonce, I':index) = getmess(sig) in
                if hconf'=hconf && n'=n && I'=I then
                event NodeTrusted(pek)
                else event NodeNotTrusted(pek);
    0.


let FogNode(pek: pekey) =
    (* sending out public key *)
    out(chan, pek);

    (* Configuration Update *)
    in(chan, (cid: configid, I:index));
    event StartingConfigurationUpdateFogNode(cid, I);
    (* Request to the Tracer *)
    out(trc, cid);
    in(trc, configData:bitstring);
    let hb = hash(configData) in
        out(pchan, (hb, I));

    (* Attestation by quote *)
    in(chan, (n': nonce, I': index));
    out(pchan, (n', I'));
    in(pchan, sig': bitstring);
    out(chan, sig');
    0.


let TPM(pek: pekey, sek: sekey, pcrSet: pcrset) =
    (* Configuration Update *)
    in(pchan, (hb: bitstring, I:index));
    event StartingConfigurationUpdateTPM(I);
    let newpcrSet = Set(pcrSet, I,
    hash((Get(pcrSet, I), hb))) in

    (* Attestation by quote *)
    in(pchan, (n': nonce, I': index));
    let hconf' = hash(Get(newpcrSet, I')) in
         out(pchan, sign((hconf', n', I'), sek));
    0.
```

```
let Tracer =
    in(trc, cid: configid);
    event TracerServingRequest();
    let configData = retrieveConfigData(cid) in
        out(trc, configData);
    0.

process
    new target: ekey;
    new pcrSet: pcrset;
    ( ORC(pk(target), pcrSet) | FogNode(pk(target)) |
    TPM(pk(target), sk(target), pcrSet) | Tracer)
```

A.2 Oblivious remote attestation

A.2.1 AK creation

```
(* Enhanced ORA protocol: Attestation Key creation *)

set ignoreTypes = false.

free chan:channel.
free tpmchan:channel.   (* Channel between the fog node and its
   tpm *)

type pkey.
type skey.
type keymat.
type result.
type keyhandle.

free tpmproof:bitstring [private].

free zeros:bitstring.
free CCpolicyauthorize:bitstring.
free CREATION:bitstring.
free TPM_GENERATED:bitstring.
free nullhandle:keyhandle.

free s1:bitstring [private].

(* Public-key encryption *)
fun penc(bitstring, pkey): bitstring.
fun pk(keymat):pkey.
fun sk(keymat):skey [private].
fun pkeyname(pkey):bitstring.
fun skeyname(skey):bitstring.
reduc forall x:bitstring, y:keymat;  pdec(penc(x, pk(y)), sk(y)) =
    x.

(* Seal and unseal *)
fun seal(skey, pkey):skey.
reduc forall m:skey, k:keymat; unseal(seal(m, pk(k)), sk(k)) = m.

(* Signatures *)
fun ok():result.
fun sign(bitstring, skey): bitstring.
reduc forall m:bitstring, k:keymat; getmess(sign(m, sk(k))) = m.
reduc forall m:bitstring, k:keymat; checksign(sign(m, sk(k)), pk(k
   )) = ok().

(* Symmetric encryption *)
```

```
fun senc(bitstring, bitstring):bitstring.
reduc forall x:bitstring, y:bitstring; sdec(senc(x, y), y) = x.

(* Cryptographic Hash *)
fun hash(bitstring): bitstring.

(* HMAC *)
fun hmac(bitstring, bitstring):bitstring.

(* Handle key *)
fun createHandle(bitstring, keyhandle, bitstring, pkey, skey):
   keyhandle.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getTemplate(createHandle(x,y,z,j,k)) = x.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getParentHandle(createHandle(x,y,z,j,k)) = y.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getPolicy(createHandle(x,y,z,j,k)) = z.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getPK(createHandle(x,y,z,j,k)) = j.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getSK(createHandle(x,y,z,j,k)) = k [private].

(* Queries *)
query attacker(s1).     (* Sanity check *)


let Orc(orcEK:keymat, ptpmEK:pkey) =
        (* Create the AK policy *)
        let hpol = hash(hash((zeros, CCpolicyauthorize, hash((
           pkeyname(pk(orcEK)),skeyname(sk(orcEK)))))))) in
        (* Create AK template *)
        new template:bitstring;
        out(chan, (hpol, template));

        (* Get the AK public part, the certificate and the
           signature over the certificate *)
        in(chan, (certinfo:bitstring, signature:bitstring, pak:
           pkey));
        if checksign(signature, ptpmEK) = ok() then
        let (objname:bitstring, magic:bitstring, magic':bitstring,
             authpol:bitstring) = certinfo in
        if objname = pkeyname(pak) then
        if magic = TPM_GENERATED then
```

```
        if magic' = template then
        if authpol = hpol then
        (* AK matches the policies *)
        out(chan, s1);

        0.

let fogNode(ekh:keyhandle, skh:keyhandle) =
        (* Get the AK policy and template *)
        in(chan, (hpol:bitstring, template:bitstring));

        (* Send TPM_create request *)
        out(tpmchan, (template, skh, hpol));
        (* Get TPM created AK *)
        in(tpmchan, (template':bitstring, skh':keyhandle, hpol':
           bitstring, pAK':pkey,
                sealedsAK':skey, hcreation':bitstring, t':
                   bitstring));

        (* Send TPM_Load request *)
        out(tpmchan, (template', skh', hpol', pAK', sealedsAK'));
        in(tpmchan, (akh:keyhandle, pakname:bitstring));

        (* Send TPM_certifyCreation request *)
        out(tpmchan, (akh, ekh, hcreation', t'));
        in(tpmchan, (certinfo:bitstring, signature:bitstring));

        (* Send all the data to the orc *)
        out(chan, (certinfo, signature, getPK(akh)));

        0.

let TPM(tpmSKh:keyhandle, tpmEKh:keyhandle) =
        (* TPM2_Create *)
        in(tpmchan, (template:bitstring, skh:keyhandle, hpol:
           bitstring));
        if skh = tpmSKh then
        (* Generate the new key *)
        new ak:keymat;
        new hcreation:bitstring;
        let t = hmac(tpmproof, (CREATION, (pkeyname(pk(ak)),
           skeyname(sk(ak))), hcreation)) in
        (* Seal AK secret part with sSK *)
        let sealedsAK = seal(sk(ak), getPK(tpmSKh)) in
        let pAK = pk(ak) in
        (* Create the wrap *)
        let wrap = (template, skh, hpol, pAK, sealedsAK, hcreation
           , t) in
        out(tpmchan, wrap);
```

```
        (* TPM2_Load *)
        in(tpmchan, (template':bitstring, skh':keyhandle, hpol':
           bitstring, pAK':pkey, sealedsAK':skey));
        if skh' = tpmSKh then
        let sAK' = unseal(sealedsAK', getSK(tpmSKh)) in
        let akh = createHandle(template', skh', hpol', pAK', sAK')
            in
        out(tpmchan, (akh, pkeyname(pAK')));

        (* TPM2_CertifyCreation *)
        in(tpmchan, (akh'':keyhandle, ekh'':keyhandle, hcreation
           '':bitstring, t'':bitstring));
        (* Get the AK key from the handler *)
        let pak'' = getPK(akh'') in
        let sak'' = getSK(akh'') in
        let xt = hmac(tpmproof, (CREATION, (pkeyname(pak''),
           skeyname(sak'')), hcreation'')) in
        if xt = t'' then        (* The key is associated to the
           right ticket and has been created inside the TPM *)
        let certinfo = (pkeyname(pak''), TPM_GENERATED,
           getTemplate(akh''), getPolicy(akh'')) in
        if ekh'' = tpmEKh then
        out(tpmchan, (certinfo, sign(certinfo, getSK(ekh''))));

        0.


process
        (* Orchestrator *)
        new orcEK:keymat;
        out(chan, pk(orcEK));

        (* TPM *)
        (* Create storage key SK *)
        new skpair:keymat;
        new sktemplate:bitstring;
        new skpolicy:bitstring;
        let tpmSKh = createHandle(sktemplate, nullhandle,skpolicy,
            pk(skpair), sk(skpair)) in
        out(chan, pk(skpair));
        (* Create Endorsement Key EK *)
        new ekpair:keymat;
        new ektemplate:bitstring;
        new ekpolicy:bitstring;
        let tpmEKh = createHandle(ektemplate, nullhandle,ekpolicy,
            pk(ekpair), sk(ekpair)) in
        out(chan, pk(skpair));
```

```
        (TPM(tpmSKh, tpmEKh) | fogNode(tpmEKh, tpmSKh) | Orc(orcEK
            , getPK(tpmEKh)))
```

A.2.2 NV-PCR attach

```
(* Attach and detach nv-PCR *)

free chan:channel.
free tpmchan:channel.

type keymat.
type skey.
type pkey.
type nvpcrhandle.
type keyhandle.
type result.

free nullhandle:keyhandle.
free nulltemplate:bitstring.
free nullpolicy:bitstring.
free zeros:bitstring.
free CCpolicyauthorize:bitstring.
free CCpolicycommandcode:bitstring.
free CCnvundefinespacespecial:bitstring.
free TPM_GENERATED:bitstring.
free WRITTEN:bitstring.

free s1:bitstring [private].
free s2:bitstring [private].

(* Asymmetric keys management *)
fun pk(keymat):pkey.
fun sk(keymat):skey [private].
fun pkeyname(pkey):bitstring.
fun skeyname(skey):bitstring.

(* Signatures *)
fun ok():result.
fun sign(bitstring, skey): bitstring.
reduc forall m:bitstring, k:keymat; getmess(sign(m, sk(k))) = m.
reduc forall m:bitstring, k:keymat; checksign(sign(m, sk(k)), pk(k
    )) = ok().

(* Cryptographic Hash *)
fun hash(bitstring): bitstring.

(* HMAC *)
fun hmac(bitstring, bitstring):bitstring.
```

```
(* Handle nv-PCR: (index, content, policy, template) *)
fun createNVPCR(bitstring, channel, bitstring, bitstring):
   nvpcrhandle.
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRIndex(createNVPCR(x,c,y,z)) = x.
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRContentChannel(createNVPCR(x,c,y,z)) = c [private].
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRPolicy(createNVPCR(x,c,y,z)) = y [private].
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRTemplate(createNVPCR(x,c,y,z)) = z [private].

(* Handle key *)
fun createHandle(bitstring, keyhandle, bitstring, pkey, skey):
   keyhandle.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyTemplate(createHandle(x,y,z,j,k)) = x.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyParentHandle(createHandle(x,y,z,j,k)) = y.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyPolicy(createHandle(x,y,z,j,k)) = z.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getPK(createHandle(x,y,z,j,k)) = j.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getSK(createHandle(x,y,z,j,k)) = k [private].

(* Queries *)
query attacker(s1).
query attacker(s2).

let Orc(orcEKh:keyhandle, ptpmEK:pkey) =
        (* Build request to attach a new nv-pcr *)

        let hpol = hash((hash(hash((zeros, CCpolicyauthorize, (
           pkeyname(getPK(orcEKh)), skeyname(getSK(orcEKh)))))),
                 CCpolicycommandcode, CCnvundefinespacespecial))
                    in
        new tpl:bitstring;
        new idx:bitstring;
        let IV = zeros in
        out(chan, (idx, tpl, hpol, IV));

        in(chan, (certInfo:bitstring, signature:bitstring));
```

```
        if checksign(signature, ptpmEK) = ok() then
        let (magic:bitstring, nvcontent:bitstring, objname:
           bitstring) = certInfo in
        if magic = TPM_GENERATED then
        if nvcontent = hash((zeros, IV)) then
        if objname = hash((tpl, WRITTEN, idx, hpol)) then
        (* The created NV-PCR matches what has been required *)
        out(chan, s1);

        0.

let fogNode(tpmEKh:keyhandle) =
        (* Attach new nv-PCR request *)
        in(chan, (idx:bitstring, tpl:bitstring, hpol:bitstring, IV
           :bitstring));

        (* TPM_NVDefineSpace request *)
        out(tpmchan, (idx, tpl, hpol));
        in(tpmchan, nvpcr:nvpcrhandle);

        (* TPM_NVExtend request *)
        out(tpmchan, (nvpcr, IV));

        (* TPM_NVCertify request *)
        out(tpmchan, (nvpcr, tpmEKh));
        in(tpmchan, (certInfo:bitstring, signature:bitstring));

        out(chan, (certInfo, signature));


        0.

let TPM(tpmEKh:keyhandle) =
        (* TPM_NVDefineSpace *)
        in(tpmchan, (idx:bitstring, tpl:bitstring, hpol:bitstring)
           );
        new nvpcrcontchan:channel;
        (* nv content initialized to zeros *)
        out(nvpcrcontchan, zeros);
        let nvprc = createNVPCR(idx, nvpcrcontchan, hpol, tpl) in
        out(tpmchan, nvprc);

        (* TPM_NVExtend *)
        in(tpmchan, (nvpcr':nvpcrhandle, IV':bitstring));
        let nvpcrcontchan' = getPCRContentChannel(nvpcr') in
        (* Extend the value inside the PCR *)
        in(nvpcrcontchan', oldvalue:bitstring);
        out(nvpcrcontchan', hash((oldvalue, IV')));
```

```
        (* TPM_NVCertify *)
        in(tpmchan, (nvpcr'':nvpcrhandle, ekh:keyhandle));
        if ekh = tpmEKh then
        let sEK = getSK(ekh) in
        let tpl'' = getPCRTemplate(nvpcr'') in
        let hpol'' = getPCRPolicy(nvpcr'') in
        let idx'' = getPCRIndex(nvpcr'') in
        let nvpcrcontchan'' = getPCRContentChannel(nvpcr'') in
        in(nvpcrcontchan'', content'':bitstring);
        let certInfo = (TPM_GENERATED, content'', hash((tpl'',
           WRITTEN, idx'', hpol''))) in
        out(tpmchan, (certInfo, sign(certInfo, sEK)));

        0.


process
        (* Orchestrator EK *)
        new orcekpair:keymat;
        let orcEKh = createHandle(nulltemplate, nullhandle,
           nullpolicy, pk(orcekpair), sk(orcekpair)) in
        out(chan, pk(orcekpair));

        (* TPM EK *)
        new tpmekpair:keymat;
        let tpmEKh = createHandle(nulltemplate, nullhandle,
           nullpolicy, pk(tpmekpair), sk(tpmekpair)) in
        out(chan, pk(tpmekpair));

        ( TPM(tpmEKh) | Orc(orcEKh, pk(tpmekpair)) | fogNode(
           tpmEKh) )
```

A.2.3 NV-PCR detach

```
(* Attach and detach nv-PCR *)

free chan:channel.
free tpmchan:channel.

type keymat.
type skey.
type pkey.
type nvpcrhandle.
type keyhandle.
type result.

free nullhandle:keyhandle.
free nulltemplate:bitstring.
```

```
free nullpolicy:bitstring.
free zeros:bitstring.
free CCpolicyauthorize:bitstring.
free CCpolicysigned:bitstring.
free CCpolicycommandcode:bitstring.
free CCnvundefinespacespecial:bitstring.
free TPM_GENERATED:bitstring.
free WRITTEN:bitstring.
free START_POLICY_SESSION:bitstring.
free VERIFIED:bitstring.

free tpmproof:bitstring [private].

free s1:bitstring [private].
free s2:bitstring [private].

(* Asymmetric keys management *)
fun pk(keymat):pkey.
fun sk(keymat):skey [private].
fun pkeyname(pkey):bitstring.
fun skeyname(skey):bitstring.

(* Signatures *)
fun ok():result.
fun sign(bitstring, skey): bitstring.
reduc forall m:bitstring, k:keymat; getmess(sign(m, sk(k))) = m.
reduc forall m:bitstring, k:keymat; checksign(sign(m, sk(k)), pk(k
   )) = ok().

(* Cryptographic Hash *)
fun hash(bitstring): bitstring.

(* HMAC *)
fun hmac(bitstring, bitstring):bitstring.

(* Handle nv-PCR: (index, content, policy, template) *)
fun createNVPCR(bitstring, channel, bitstring, bitstring):
   nvpcrhandle.
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRIndex(createNVPCR(x,c,y,z)) = x.
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRContentChannel(createNVPCR(x,c,y,z)) = c [private].
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRPolicy(createNVPCR(x,c,y,z)) = y [private].
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRTemplate(createNVPCR(x,c,y,z)) = z [private].

(* Handle key *)
```

```
fun createHandle(bitstring, keyhandle, bitstring, pkey, skey):
   keyhandle.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyTemplate(createHandle(x,y,z,j,k)) = x.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyParentHandle(createHandle(x,y,z,j,k)) = y.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyPolicy(createHandle(x,y,z,j,k)) = z.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getPK(createHandle(x,y,z,j,k)) = j.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getSK(createHandle(x,y,z,j,k)) = k [private].

(* Queries *)
query attacker(s1).
query attacker(s2).

let Orc(orcEKh:keyhandle, ptpmEK:pkey, nvpcrh:nvpcrhandle) =
        (* Algorithm 1: auth nv index deletion *)
        in(chan, nonce:bitstring);
        let idx = getPCRIndex(nvpcrh) in
        let hpol = hash(hash((zeros, CCpolicysigned, pkeyname(
           getPK(orcEKh))))) in
        let hhpol = hash(hpol) in
        let signhhpol = sign(hhpol, getSK(orcEKh)) in
        let hcp = hash((CCnvundefinespacespecial, idx)) in
        let ahash = hash((nonce, zeros, hcp)) in
        let signahash = sign(ahash, getSK(orcEKh)) in

        out(chan, (nvpcrh, hcp, signahash, hpol, hhpol, signhhpol)
           );

        0.

let fogNode(tpmEKh:keyhandle, orcEKh:keyhandle) =
        (* Start policy session request *)
        out(tpmchan, START_POLICY_SESSION);
        in(tpmchan, nonce:bitstring);
        out(chan, nonce);

        (* Receive REQ delete *)
        in(chan, (nvpcrh:nvpcrhandle, hcp:bitstring, signahash:
           bitstring, hpol:bitstring,
                hhpol:bitstring, signhhpol:bitstring));
```

```
        (* TPM_VerifySignature request *)
        out(tpmchan, (hhpol, signhhpol, orcEKh));
        in(tpmchan, t:bitstring);

        (* TPM_PolicySigned request *)
        out(tpmchan, (signahash, hcp, nonce, orcEKh));

        (* TPM_PolicyAuthorize request *)
        out(tpmchan, (hpol, t, pkeyname(getPK(orcEKh))));

        (* TPM_PolicyCommandCode set request *)
        out(tpmchan, CCnvundefinespacespecial);

        (* TPM_NVUndefineSpaceSpecial request *)
        out(tpmchan, nvpcrh);

        0.

let TPM(tpmEKh:keyhandle, orcEKh:keyhandle) =
        (* Start policy session *)
        in(tpmchan, command:bitstring);
        if command = START_POLICY_SESSION then
        new nonce:bitstring;
        new sessionhpol:channel;
        new sessioncphash:channel;
        out(sessionhpol, zeros);
        out(tpmchan, nonce);

        (* TPM_VerifySignature *)
        in(tpmchan, (hhpol:bitstring, signhhpol:bitstring, handle:
           keyhandle));
        if handle = orcEKh then
        if checksign(signhhpol, getPK(orcEKh)) = ok then
        if getmess(signhhpol) = hhpol then
        let t = hmac(tpmproof, (VERIFIED, hhpol, pkeyname(getPK(
           orcEKh)))) in
        out(tpmchan, t);

        (* TPM_PolicySigned *)
        in(tpmchan, (signahash':bitstring, hcp':bitstring, nonce':
           bitstring, handle':keyhandle));
        if handle' = orcEKh then
        let ahash = hash((nonce', zeros, hcp')) in
        if checksign(signahash', getPK(orcEKh)) = ok then
        if getmess(signahash') = ahash then
        in(sessionhpol, oldsessionhpol':bitstring);
        out(sessionhpol, hash(hash((oldsessionhpol',
           CCpolicysigned, pkeyname(getPK(orcEKh))))));
```

```
        out(sessioncphash, hcp');

        (* TPM_PolicyAuthorize *)
        in(tpmchan, (hpol'':bitstring, t'':bitstring, orcname'':
           bitstring));
        in(sessionhpol, oldsessionhpol'':bitstring);
        if oldsessionhpol'' = hpol'' then
        let newt = hmac(tpmproof, (VERIFIED, hash(hpol''), orcname
           '')) in
        if newt = t'' then
        out(sessionhpol, hash((zeros, CCpolicyauthorize, orcname
           '')));

        (* TPM_PolicyCommandCode *)
        in(tpmchan, commandcode:bitstring);
        let sessioncc = commandcode in
        in(sessionhpol, oldsessionhpol''':bitstring);
        out(sessionhpol, hash((oldsessionhpol''',
           CCpolicycommandcode, CCnvundefinespacespecial)));

        (* TPM_NVUndefineSpaceSpecial *)
        in(tpmchan, nvpcrh:nvpcrhandle);
        if sessioncc = CCpolicycommandcode then
        in(sessionhpol, hpolvalue:bitstring);
        if hpolvalue = getPCRPolicy(nvpcrh) then
        in(sessioncphash, cphashvalue:bitstring);
        if cphashvalue = hash((CCnvundefinespacespecial,
           getPCRIndex(nvpcrh))) then
        (* Deletion can be done *)
        out(chan, s1);

        0.


process
        (* Orchestrator EK *)
        new orcekpair:keymat;
        let orcEKh = createHandle(nulltemplate, nullhandle,
           nullpolicy, pk(orcekpair), sk(orcekpair)) in
        out(chan, pk(orcekpair));

        (* TPM EK *)
        new tpmekpair:keymat;
        let tpmEKh = createHandle(nulltemplate, nullhandle,
           nullpolicy, pk(tpmekpair), sk(tpmekpair)) in
        out(chan, pk(tpmekpair));

        (* Create nv register *)
        new template:bitstring;
```

```
        let policy = hash((hash(hash((zeros, CCpolicyauthorize, (
            pkeyname(getPK(orcEKh)), skeyname(getSK(orcEKh)))))),
                  CCpolicycommandcode, CCnvundefinespacespecial))
                     in
        new index:bitstring;
        new nvcontent:channel;
        let nvpcrh = createNVPCR(index, nvcontent, policy,
            template) in

        ( TPM(tpmEKh, orcEKh) | Orc(orcEKh, pk(tpmekpair), nvpcrh)
             | fogNode(tpmEKh, orcEKh) )
```

A.2.4 Measurement update request

```
free chan:channel.
free tpmchan:channel.

type keymat.
type skey.
type pkey.
type pcrhandle.
type keyhandle.
type result.

free nullhandle:keyhandle.
free nulltemplate:bitstring.
free nullpolicy:bitstring.
free zeros:bitstring.
free pcr_iv:bitstring [private].
free CCpolicypcr:bitstring.
free VERIFIED:bitstring.
free authHash:bitstring.
free CCpcrextend:bitstring.
free success:bitstring.

free tpmproof:bitstring [private].
free hk_orc_aagt:bitstring [private].
free fqpn:bitstring [private].

free s1:bitstring [private].
free s2:bitstring [private].

(* Asymmetric keys management *)
fun pk(keymat):pkey.
fun sk(keymat):skey [private].
fun pkeyname(pkey):bitstring.
fun skeyname(skey):bitstring.
```

```
(* Signatures *)
fun ok():result.
fun sign(bitstring, skey): bitstring.
reduc forall m:bitstring, k:keymat; getmess(sign(m, sk(k))) = m.
reduc forall m:bitstring, k:keymat; checksign(sign(m, sk(k)), pk(k
   )) = ok().

(* Tracer *)
fun tracer(bitstring): bitstring [private].

(* Cryptographic Hash *)
fun hash(bitstring): bitstring.

(* HMAC *)
fun hmac(bitstring, bitstring):bitstring.

(* Handle PCR: (index, content, policy, template) *)
fun createPCR(bitstring, channel, bitstring, bitstring):pcrhandle.
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRIndex(createPCR(x,c,y,z)) = x.
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRContentChannel(createPCR(x,c,y,z)) = c [private].
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRPolicy(createPCR(x,c,y,z)) = y [private].
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRTemplate(createPCR(x,c,y,z)) = z [private].

(* Handle key *)
fun createHandle(bitstring, keyhandle, bitstring, pkey, skey):
   keyhandle.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyTemplate(createHandle(x,y,z,j,k)) = x.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyParentHandle(createHandle(x,y,z,j,k)) = y.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyPolicy(createHandle(x,y,z,j,k)) = z.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getPK(createHandle(x,y,z,j,k)) = j.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getSK(createHandle(x,y,z,j,k)) = k [private].


(* Queries *)
query attacker(s1).
```

```
query attacker(s2).


let Orc(orcEKh:keyhandle, ptpmEK:pkey, pcrh:pcrhandle) =
    let hupdate = hmac(hk_orc_aagt, hash(tracer(fqpn))) in
    (* Algorithm 2 *)
    let pcrcontent = getPCRContentChannel(pcrh) in
    in(pcrcontent, oldvalue:bitstring);
    let newpcrvalue = hash((oldvalue, hupdate)) in
    out(pcrcontent, newpcrvalue);
    let hpol = hash((zeros, CCpolicypcr, getPCRIndex(pcrh), hash(
       newpcrvalue))) in
    let hhpol = hash(hpol) in
    let signhhpol = sign(hhpol, getSK(orcEKh)) in
    out(chan, (hpol, signhhpol));

    (* Algorithm 4 - Verify *)

    in(chan, sigAuditInfo: bitstring);
    if (checksign(sigAuditInfo, ptpmEK) = ok()) then
    let cpHash = hash((CCpcrextend, getPCRIndex(pcrh), authHash,
       hupdate)) in
    let rpHash = hash((success, CCpcrextend)) in
    let haudit = hash((zeros, cpHash, rpHash)) in
    if(getmess(sigAuditInfo) = haudit) then
    out(chan, s1);
    0.



let fogNode(ptpmEK:pkey, porcEK:pkey) =
    in(chan, (hpol:bitstring, signhhpol:bitstring));
    out(tpmchan, (hpol, signhhpol));

    in(tpmchan, t:bitstring);
    new SessionType_HMAC:bitstring;
    out(tpmchan, SessionType_HMAC);

    in(tpmchan, Hhs:bitstring);
    new AUDIT:bitstring;
    let hfqpn = hmac(hk_orc_aagt, hash(tracer(fqpn))) in
        out(tpmchan, (hfqpn, AUDIT));

    in(tpmchan, sigAuditInfo:bitstring);
    out(chan, sigAuditInfo);
    0.


let TPM(tpmEKh:keyhandle, orcEKh:keyhandle, pcrh:pcrhandle) =
```

```
    in(tpmchan, (hpol:bitstring, signhhpol:bitstring));

    (* TPM2_VerifySignature *)

    if checksign(signhhpol, getPK(orcEKh)) = ok() then
        if hash(hpol) = getmess(signhhpol) then
            let t = hmac(tpmproof, (VERIFIED, hash(hpol), pkeyname
               (getPK(orcEKh)))) in
            out(tpmchan, t);

    (* TPM2_StartAuthSession *)

    in(tpmchan, SessionType_HMAC:bitstring);
    new Hhs: bitstring;
    out(tpmchan, Hhs);

    (* TPM2_PCR_Extend *)

    in(tpmchan, (hfqpn:bitstring, AUDIT:bitstring));

    let pcrcontent = getPCRContentChannel(pcrh) in
    in(pcrcontent, oldvalue:bitstring);
    let newpcrvalue = hash((oldvalue, hfqpn)) in
    out(pcrcontent, newpcrvalue);
    (* Algorithm 3 *)
    let cpHash = hash((CCpcrextend, getPCRIndex(pcrh), authHash,
       hfqpn)) in
    let rpHash = hash((success, CCpcrextend)) in
    let haudit = hash((zeros, cpHash, rpHash)) in
    (* TPM2_GetSessionAuditDigest *)
    let sigAuditInfo = sign(haudit, getSK(tpmEKh)) in
    out(tpmchan, sigAuditInfo);
    0.



process
    (* Orchestrator EK *)
    new orcekpair:keymat;
    let orcEKh = createHandle(nulltemplate, nullhandle, nullpolicy
       , pk(orcekpair), sk(orcekpair)) in
    out(chan, pk(orcekpair));

    (* TPM EK *)
    new tpmekpair:keymat;
    let tpmEKh = createHandle(nulltemplate, nullhandle, nullpolicy
       , pk(tpmekpair), sk(tpmekpair)) in
    out(chan, pk(tpmekpair));
```

```
    (* Create pcrs *)
    new index:bitstring;
    new pcrcontent:channel;
    out(pcrcontent, pcr_iv);
    let pcrh = createPCR(index, pcrcontent, nullpolicy,
       nulltemplate) in

    ( TPM(tpmEKh, orcEKh, pcrh) | Orc(orcEKh, pk(tpmekpair), pcrh)
        | fogNode(pk(tpmekpair), pk(orcekpair)) )
```

A.2.5 ORA

```
free chan:channel.
free tpmchan:channel [private].

type keymat.
type skey.
type pkey.
type keyhandle.
type pcr.
type nvpcrhandle.
type sessionhandle.
type result.

free nullhandle:keyhandle.
free nulltemplate:bitstring.
free nullpolicy:bitstring.
free zeros:bitstring.
free CCpolicyauthorize:bitstring.
free CCpolicynv:bitstring.
free CCpolicypcr:bitstring.
free START_POLICY_SESSION:bitstring.
free VERIFIED:bitstring.

free tpmproof:bitstring [private].
free ticketproof:bitstring [private].

free s1:bitstring [private].
free s2:bitstring [private].

(* Asymmetric keys management *)
fun pk(keymat):pkey.
fun sk(keymat):skey [private].
fun pkeyname(pkey):bitstring.
fun skeyname(skey):bitstring.

(* Signatures *)
```

```
fun ok():result.
fun sign(bitstring, skey): bitstring.
reduc forall m:bitstring, k:keymat; getmess(sign(m, sk(k))) = m.
reduc forall m:bitstring, k:keymat; checksign(sign(m, sk(k)), pk(k
   )) = ok().

(* Cryptographic Hash *)
fun hash(bitstring): bitstring.

(* HMAC *)
fun hmac(bitstring, bitstring):bitstring.

(* Handle nv-PCR: (index, content, policy, template) *)
fun createNVPCR(bitstring, channel, bitstring, bitstring):
   nvpcrhandle.
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRIndex(createNVPCR(x,c,y,z)) = x.
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRContentChannel(createNVPCR(x,c,y,z)) = c [private].
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRPolicy(createNVPCR(x,c,y,z)) = y [private].
reduc forall x:bitstring, c:channel, y:bitstring, z:bitstring;
        getPCRTemplate(createNVPCR(x,c,y,z)) = z [private].

fun nvpcrhandlename(nvpcrhandle):bitstring.

(* Handle key *)
fun createHandle(bitstring, keyhandle, bitstring, pkey, skey):
   keyhandle.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyTemplate(createHandle(x,y,z,j,k)) = x.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyParentHandle(createHandle(x,y,z,j,k)) = y.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getKeyPolicy(createHandle(x,y,z,j,k)) = z.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getPK(createHandle(x,y,z,j,k)) = j.
reduc forall x:bitstring, y:keyhandle, z:bitstring, j:pkey, k:skey
   ;
        getSK(createHandle(x,y,z,j,k)) = k [private].

(* Handle Session : (Session Type, Policy Digest) *)
fun createSession(bitstring, channel):sessionhandle.
reduc forall x:bitstring, y:channel;
    getSessionType(createSession(x,y)) = x.
```

```
reduc forall x:bitstring, y:channel;
    getPolicyDigest(createSession(x,y)) = y.

(* Queries *)
query attacker(s1).
query attacker(s2).


let X(ypAK:pkey) =
    new n:bitstring;
    out(chan, n);
    in(chan, sig:bitstring);
    if(checksign(sig, ypAK) = ok()) then
        out(chan, s1);

    0.

let Y(P:bitstring, T:bitstring, AKh:keyhandle, porcEK:pkey) =
    in(chan, n:bitstring);

    out(tpmchan, START_POLICY_SESSION);
    in(tpmchan, Hps:sessionhandle);

    out(tpmchan, (P, T, pkeyname(porcEK)));

    out(tpmchan, (n, AKh));

    in(tpmchan, sig:bitstring);
    out(chan, sig);

    0.


let TPM(pcrcontent:bitstring, nvpcrh: nvpcrhandle) =
    (* TPM2_StartAuthSession *)
    in(tpmchan, policysession:bitstring);
    new hpolchan:channel;
    out(hpolchan, zeros);
    let Hps = createSession(policysession, hpolchan) in
    out(tpmchan, Hps);

    (* TPM2_PolicyNV *)

    in(getPolicyDigest(Hps), emptyhpol:bitstring);
    in(getPCRContentChannel(nvpcrh), nvpcr:bitstring);
    let args = (nvpcr, zeros) in
    let nvhpol = hash((emptyhpol, CCpolicynv, args,
       nvpcrhandlename(nvpcrh))) in
    out(getPolicyDigest(Hps), nvhpol);
```

```
    (* TPM2_PolicyPCR *)

    in(getPolicyDigest(Hps), nvhpol':bitstring);
    let pcrhpol = hash((nvhpol', CCpolicypcr, pcrcontent, hash(
       pcrcontent))) in
    out(getPolicyDigest(Hps), pcrhpol);

    (* TPM2_PolicyAuthorize *)

    in(tpmchan, (P:bitstring, T:bitstring, porcEKname:bitstring));
    in(getPolicyDigest(Hps), hpol:bitstring);
    let t = hmac(ticketproof, (VERIFIED, hash(hpol), porcEKname))
       in
    if( (P = hpol) && (T = t) ) then
        out(getPolicyDigest(Hps), zeros);
        let newhpol = hash((hpol, CCpolicyauthorize, porcEKname))
           in
        out(getPolicyDigest(Hps), newhpol);

        in(tpmchan, (n:bitstring, AKh:keyhandle));
        if(newhpol = getKeyPolicy(AKh)) then
            let sig = sign(n, getSK(AKh)) in
            out(tpmchan, sig);
    0.



process
    (* Orchestrator EK *)
    new orcekpair:keymat;
    let orcEKh = createHandle(nulltemplate, nullhandle, nullpolicy
       , pk(orcekpair), sk(orcekpair)) in
    out(chan, pk(orcekpair));

    (* Y Attestation Key *)
    new akpair: keymat;

    (* Create PCRs *)
    new pcrcontent:bitstring;

    (* Create nv register *)
    new template:bitstring;
    new index:bitstring;
    new nvcontent:channel;
    new nvpcr:bitstring;
    out(nvcontent, nvpcr);
```

```
    let nvpcrh = createNVPCR(index, nvcontent, nullpolicy,
        template) in

let args = (nvpcr, zeros) in
let nvstep = hash((zeros, CCpolicynv, args, nvpcrhandlename(
    nvpcrh))) in
let pcrstep = hash((nvstep, CCpolicypcr, pcrcontent, hash(
    pcrcontent))) in
let P = pcrstep in

let T = hmac(ticketproof, (VERIFIED, hash(P), pkeyname(getPK(
    orcEKh)))) in
let authPol = hash((P, CCpolicyauthorize, pkeyname(getPK(
    orcEKh)))) in
let AKh = createHandle(nulltemplate, nullhandle, authPol, pk(
    akpair), sk(akpair)) in

( !X(getPK(AKh)) | !Y(P, T, AKh, getPK(orcEKh)) | !TPM(
    pcrcontent, nvpcrh) )
```