POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Formal verification of Remote Attestation protocols in a Fog Computing Architecture

Supervisors

Prof. Riccardo Sisto
Dott. Fulvio Valenza
Dott. Simone Bussa

Candidate

Andreina Erika Ricci

July 2022

Acknowledgments

I would like to thank all my supervisors, for their guidance and help.

I would also like to thank all the people who put up with me during these years.

To Andrea. You were always by my side, your love and support have meant more to me than you could possibly realize.

To Anna. Thank you for always having time for me and understanding me. I couldn't have wished for a better friend.

Finally, to my whole family. Thank you for the unconditional love, patience and support during my academic path. Giorgia, thank you for being a great sister.

Contents

1 Introduction **10**
1.1 Structure of the document 10

2 General concepts **12**
2.1 Remote attestation 12
2.1.1 Architecture 12
2.2 Security properties 13
2.3 Security analysis 14
2.3.1 Formal verification 14
2.4 Tamarin prover framework 15
2.4.1 State, Facts, Rules 15
2.4.2 Lemmas 17
2.4.3 Restrictions 17
2.4.4 Channels 19

3 Remote attestation protocols **20**
3.1 Attestation by quote 21
3.1.1 Update protocol 21
3.1.2 Attestation by quote protocol 21
3.2 The Oblivious Remote Attestation 22
3.2.1 Attestation Key creation 23
3.2.2 Measurement Update protocol 26
3.2.3 ORA protocol 28
3.2.4 NVPCRs administration 30

4 Objective of the thesis **34**
4.1 Objective of the Thesis 34

5 Design **36**
5.1 Attestation by quote 36
5.1.1 Update protocol model 37
5.1.2 Attestation by quote protocol model 39
5.2 Oblivious Remote Attestation 40
5.2.1 Attestation Key creation protocol model 41
5.2.2 Measurement update protocol 45
5.2.3 Inter-Vfs Oblivious Remote Attestation protocol model 47
5.2.4 Attaching NVPCRs protocol model 52
5.2.5 Detatching NVPCRs protocol model 55

6 Results **60**
6.1 Attestation by quote verification . . . 60
6.1.1 Attestation by quote: Lemmas . . . 60
6.1.2 Attestation by quote: Results . . . 62
6.2 Oblivious Remote Attestation verification . . . 65
6.2.1 Attestation Key creation: Results . . . 65
6.2.2 Measurement update: Results . . . 69
6.2.3 Oblivious Remote Attestation: Results . . . 72
6.2.4 Attaching NVPCRs: Results . . . 75
6.2.5 Detaching NVPCRs: Results . . . 77

7 Conclusions **80**
7.1 Conclusion . . . 80
7.2 Future implementations . . . 81

List of Figures

2.1 Fog Computing Architecture. 13

3.1 Update Measurement protocol. 21
3.2 Attestation by Quote protocol. 22
3.3 Oblivious-based inter-VF Remote Attestation. 23
3.4 Attestation Key creation protocol. 25
3.5 Measurement Update protocol. 27
3.6 Inter-VFs Oblivious Remote Attestation protocol. 29
3.7 Attaching NVPCRs protocol. 31
3.8 Detaching NVPCRs protocol. 33

6.1 Attestation by Quote protocol - Attacker coercing id of the configuration file and index. 63
6.2 Attestation by Quote protocol - Attacker coercing the hash of the configuration file. 63
6.3 Attestation by Quote protocol - Attacker coercing nonce. 64
6.4 Attestation by Quote protocol - Attacker coercing the signing message. 64
6.5 Attestation by Quote - Aliveness and Injective agreement result. . . . 65
6.6 Attestation Key creation protocol - Message Authentication result. . 66
6.7 Attestation Key creation protocol - Aliveness and Injective agreement. 68
6.8 Measurement Update protocol - Message authentication - 1. 70
6.9 Measurement Update protocol - Message authentication - 2. 71
6.10 Measurement Update protocol - Aliveness. 72
6.11 ORA protocol - Aliveness and Injective agreement. 73
6.12 ORA protocol - Message authentication. 74
6.13 Attaching NVPCRs protocol - Message Authentication - 1. 76
6.14 Attaching NVPCRs protocol - Message Authentication - 1. 77
6.15 Attaching NVPCRs protocol - Aliveness and Injective agreement. . . 77
6.16 Detaching NVPCRs protocol - Message authentication. 78
6.17 Detaching NVPCRs protocol - Aliveness and Injective agreement. . . 79

List of Tables

3.1 Table of abbreviations. 20

Chapter 1

Introduction

The purpose of this thesis is to model and verify Remote Attestation protocols in a fog computing environment. Fog computing is a cloud infrastructure that consists in devices connecting to a cloud, but instead of forwarding data to the center of the cloud, much of the processing is done at the edges.

This work is related to the European project called Rainbow. Rainbow [14] is a trusted fog computing platform that provides simple management of heterogeneous IoT (Internet of things) devices, which contain sensors and exchange data over the Internet.

In a fog computing infrastructure, there can be different security issues [13]. Since a great number of devices is involved in the architecture, there are concerning issues about authentication, trust and privacy. It is important to establish trust between all the components, because they all interact with each other through different networks: this kind of flexibility can extend the attack surface. In this scenario, a malicious agent can take control of one of the devices and compromise the entire system. In order to avoid this issue, protocols of remote attestation are needed. Starting from interaction between two systems (Fog Node and Orchestrator), RAINBOW provided two important functionalities in its attestation Toolkit: Attestation by Proof and Attestation by Quote. Both these functionalities are for automatic establishment of trust through the verification of the integrity of the fog nodes. The remote attestation process, in general, allows to understand if a fog node can be considered trusted, verifying its correctness. For this reason it is important that these protocols are secure. Security properties of a protocol can be evaluated using different techniques. In this thesis, the correctness of the remote attestation protocols is analysed using the Formal Verification, which is an a-priori security verification of a mathematical model.

The tool used for this purpose is the Tamarin Prover, which takes as input a symbolic model and automatically proves its correctness.

1.1 Structure of the document

The thesis document is structured in seven chapters, organized as follows:

- **Chapter 2:** it presents the general concepts and the Tamarin Prover;

- **Chapter 3:** it describes the remote attestation protocols;
- **Chapter 4:** it describes the objective of this work of thesis;
- **Chapter 5:** it describes the implementation of the Tamarin models;
- **Chapter 6:** it shows the results of the model's verification;
- **Chapter 7:** it presents the conclusions.

Chapter 2

General concepts

2.1 Remote attestation

The remote attestation [8] is a method used to verify the integrity of a software that is running on a remote device. The two agents involved in the attestation process are the Prover and the Verifier: the Verifier is the agent that wants to verify the integrity of the Prover, which has to provide a verifiable evidence. The data sent by the Prover to the Verifier have to be authentic. In order to provide authenticity a trust anchor has to be used. In this work, the trust anchor part of the reference architecture is the TPM (Trusted Platform Module).

2.1.1 Architecture

The reference architecture considered in this fog computing scenario, is based on three main actors:

- Orchestrator (Orc): it is a coordinator component, that handles the life-cycle of all the edge-computing architecture, managing also applications and network services;
- Fog Node: it is one of the edge devices with some ability to process the collected data from sensors;
- TPM (Trusted Platform Module)[5]: it is a module present in the fog nodes, a secured cryptographic co-processor which can communicate with its host. The TPM capabilities are[10]:
  - Secure generation of three hierarchies of keys: Platform(for platform's firmware), Endorsement(for privacy-sensitive operations), Storage(for the platform's owner);
  - Hardware random number generator;
  - Binding: data encryption with the storage key;
  - Sealing: similar to binding, but with a specific TPM state;
  - Authentication of hardware with the Endorsement Key (EK).

  The TPM contains a set of specific memory registers used to measure the state and the configuration of the software, they are called PCRs (Platform Configuration Registers).

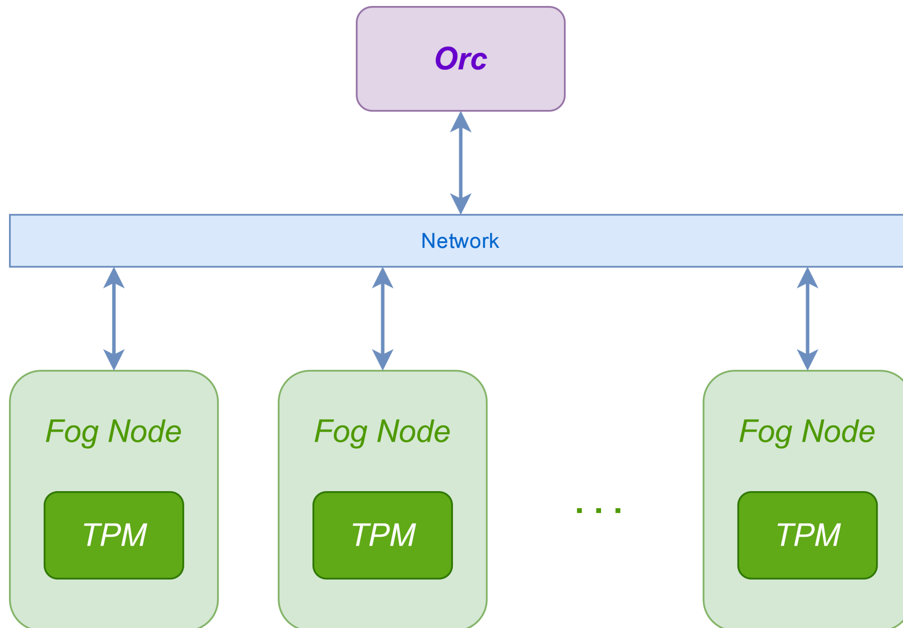

Figure 2.1: Fog Computing Architecture.

In this environment, is important that the nodes who join the cluster are trusted, in order to guarantee a certain level of security. Considering this architecture, for the purpose of remote attestation, the fog node is identified as the Prover, while the Orchestrator is the Verifier.

When a node wants to join the cluster, it has to provide a verifiable evidence on its configuration integrity, proving to the Orchestrator that its state is correct.

2.2 Security properties

A secure system, in general, has to follow the principles of the CIA model:

- Confidentiality: information should be hidden from unauthorized entities, for example using cryptographic operations;
- Integrity: data should not be manipulated and changed from the original ones;
- Availability: data should be always accessible to authorized entities.

In particular, a remote protocol should guarantee that some specific security properties are maintained in the execution. The main security properties analysed in this thesis are:

- Secrecy: it is a property that specifies the need for certain data to not be discoloured, for example symmetric and private asymmetric keys;
- Authentication: provides proof of authenticity of messages and agents interacting during the execution of the protocol;
- Aliveness: it is a form of authentication that guarantees to an agent A aliveness of another agent B if, whenever A completes a run of the protocol, then B has previously been running the protocol;

- Injective Agreement: it is a stronger authentication property that guarantees to an agent A a unique matching partner B in the running of the protocol, in order to prevent replay attacks.

2.3 Security analysis

A system is considered secure if it guarantees some specific security properties and a certain level of security [12]. In order to verify if a system is secure, a security analysis is needed. Different techniques[16] can be used:

- Vulnerability assessment: identification of network vulnerabilities in a system;
- Penetration testing: attempt to exploit vulnerabilities found in a system;
- Code analysis: style checking, documentation checking, code checking;
- Formal verification [11]: The formal verification is the static analysis of a system formal model, that is a mathematically-based model;
- Auditing: meetings for evaluating security.

2.3.1 Formal verification

This work is based on the verification of symbolic models of remote attestation protocols. The symbolic analysis approach uses models which refer to the Dolev-Yao(DY) model [7], [9]. The DY model is based on some assumptions:

- Abstract data types: data are symbolic terms;
- Perfect cryptography: cryptographic primitives are perfect black-boxes with ideal properties;
- The attacker has full control over the network: it can read, delete, substitute, build and insert messages and can also execute cryptographic operations.

There are different tools for formal verification of security protocols in the symbolic model, the main ones are:

- ProVerif: it is a tool for automatic verification of cryptographic protocols [4], it can verify protocols for an unbounded number of sessions and the abstractions used make the verification sound but not always complete. This tool has been used, for example, for a formal analysis to verify security properties of procedures executed when a mobile device changes LTE cell or switches to a UMTS network [6];
- Tamarin Prover: The Tamarin Prover is a modern alternative to ProVerif, it also provides an interactive mode that can be used to explore the states and guide tool to the proof if it does not complete. The Tamarin Prover has been used, for example, for formal analysis of an ECC-DAA, Direct Anonymous Attestation [17].

2.4 Tamarin prover framework

The protocols analysed in this thesis are modeled and verified using the Tamarin Prover. Tamarin [15] is a tool used for symbolic analysis of security protocols in presence of a Dolev-Yao (DY) style network adversary (Section 2.3.1).
The Tamarin prover allows to analyse security protocols starting from a symbolic model and automatically constructing proofs that the protocol satisfies its security properties. The models of the protocols are based on multiset rewriting rules and the security properties are written using first-order logic, which is an extension of propositional logic where the atomic propositions are replaced by predicates and new concepts, such as quantifiers, functions and variables. The model taken as input specifies the agents, their roles and their actions, as well as the adversary and the security properties to verify, that can be modeled as trace-properties or as observational equivalence of two transition systems. The adversary can interact with the protocol by sending or manipulating messages on the network.

In order to analyse the protocol, Tamarin constructs proofs, based on heuristics, and counterexamples, representing the possible attacks. The correctness of a protocol is not always a decidable problem, so Tamarin may not terminate. In this case Tamarin's interactive mode can be used to explore the states and guide Tamarin to the proof.

In the following sections, all the main characteristic that the Tamarin Prover requires in modelling the protocols will be briefly explained.

2.4.1 State, Facts, Rules

As said before, Tamarin works with symbolic models and protocols are modeled using mutilset rewriting rules. In a symbolic model all messages are modeled as terms. The rules operate on the states of the protocol execution, each represented as a finite multiset of facts, which are distinguished terms. They define how the system transitions to a new state. A rule is written as:

$$[LHS] - [actions] \rightarrow [RHS].$$

Rules are composed by finite sequences of facts called premise and conclusion, also referred to as left hand side (LHS) and right hand side (RHS), and they are separated by an arrow. In the premise there are the facts that have to be present in the current state for the rule to be executed, after the execution the facts in the conclusion will be added to the state. Actions between LHS and RHS are associated to the specific rule and represent labels of it. During the execution, actions yield the traces, that are then captured as lemmas.

The following example (2.4.1) shows how two agents interact with each other.

Example 2.4.1. Alice wants to send an encrypted nonce to Bob, Bob will decrypt the message with its private key and do an hash of the nonce, then send it back to Alice.

```
theory first_example
begin
```

```
builtins: hashing, asymmetric-encryption

rule Register_pk:
    [
        Fr(~ltkB)
    ]
  -->
    [
        !Ltk($B, ~ltkB)
        , !Pk($B, pk(~ltkB))
        , Out(pk(~ltkB))
    ]

rule Alice_send:
    [
        Fr(~nonce)
        , !Pk($B, pkB)
    ]
    --[Send($A, ~nonce)]->
    [
         Out( aenc(~nonce, pkB) )
    ]

rule Bob_receive:
    [
        !Ltk($B, ltkB),
        In(n)
    ]
    --[Send_hash($B, h(n))]->
    [
        Out( h(adec(n, ltkB)) )
    ]


rule Alice_receive:
    [
        In(hashed_nonce)
    ]
    --[Receive($A, hashed_nonce)]->
    []

end
```

In this example (2.4.1), Tamarin's built-in functions are used. The Tamarin prover has a number of built-in equational theories (a set of equations) that can be used for models, such as hashing, asymmetric-encryption, signing and diffie-hellman. Built-ins contain some useful functions and facts, in this example the used built-ins are needed for the hash function, and the encryption and decryption operations.

The facts present in this example are:

- Fr: it is a fact that models a frashly generated unique term;
- Out: it is a fact that models the protocol sending a message on the public channel;
- In: it is a fact that models the protocol receiving a message on the public channel;
- !Ltk: it associates the agent with its private key;
- !Pk: it associates the agent with its public key.

The exclamation mark "!" specifies that a fact is persistent, which means that it can be consumed more than one time, as opposed to linear ones.

2.4.2 Lemmas

The security properties of a protocol that need to be evaluated using the Tamarin prover are described by lemmas. Action facts yield traces, lemmas capture these traces and evaluate them. A lemma is proved correct if it is satisfied by all traces or if exists one trace, it depends on how it is specified. The Example 2.4.2 defines a possible lemma for the previous one (2.4.1).

Example 2.4.2. This lemma is the one that verifies that the protocol is executable and the actions follow a specific flow considering the timepoints i, j and k.

```
lemma functional_correctness:
    exists-trace
    "Ex A B m #i #j #k.
    Send(A,m)@i &
    Send_hash(B,m) @j &
    Receive(A, m) &
    i<j &
    j<k"
```

2.4.3 Restrictions

In Tamarin, it is possible to give constraints to the protocol execution using restrictions. Restrictions refer to action facts, which are specific for each rule. Given that action facts yield traces, restrictions on them limit the yielded traces in the protocol execution and they can be used to remove degenerate cases. There are numerous types of restriction, some common ones are the ones used for comparison, in particular the equality one, which is important for the verification of signatures.

In the example below (2.4.3), Alice verifies the signature on the message containing the nonce received signed by Bob. In order to verify the signature, the restriction Equality is specified, it refers to the action fact present in the "Alice_receive" rule and it allows to consider for analysis only the traces of the protocol in which the verify function is true.

Example 2.4.3. Alice wants to send a nonce to Bob, Bob will sign the message with its private key and then send it back to Alice, who will verify the signature.

```
theory restriction_example_signature
builtins: signing

rule Register_pk:
    [
        Fr(~ltkB)
    ]
  -->
    [
        !Ltk($B, ~ltkB)
        , !Pk($B, pk(~ltkB))
        , Out(pk(~ltkB))
    ]

rule Alice_send:
    [
        Fr(~nonce)
    ]
    --[Send($A, ~nonce)]->
    [
        , Out( ~nonce )
    ]

rule Bob_receive:
    [
        !Ltk($B, ~ltkB)
        In(n)
    ]
    --[Send_signed($B, n)]->
    [
        Out( <n, sign(n, ltkB)> )
    ]


rule Alice_receive:
    [
        !Pk($B, pkB)
        In(<nonce, signed_nonce>)
    ]
    --[
        Receive($A, hashed_nonce),
        Eq(verify(signed_nonce, nonce, pkB), true)
    ]->
    []
```

```
restriction Equality:
    "All x y #i. Eq(x,y) @i ==> x = y"

end
```

2.4.4 Channels

In the previous examples, the agents communicate over a public channel using the In and Out facts. The public channel, however, is totally accessible by the attacker, who can read, delete and inject messages, having also the possibility to construct messages starting from an initial knowledge. In order to manage the traffic, channels can be modeled using rules to limit attacker's control over them, making sure that they guarantee some specific security properties (confidentially, authenticity).

Different types of channels can be defined using rules, depending on the properties they have to satisfy. There are three main types of channels:

- Confidential channel: it is a channel in which the attacker can send messages, but can not read messages sent between the agents;
- Authentic channel: it is a channel in which the attacker can read the messages, but can not send or modify messages;
- Secure channel: it is a channel in which the attacker can not send, modify or read the messages, because it is both confidential and authentic. The rules for the definition of this type of channel are described in the 2.4.4 example.

Example 2.4.4. This example shows the two rules needed for the creation of a secure channel between two agents. The !Sec persistent fact is used to bind the sender, the receiver and the message. Sending a message on this type of channel, Tamarin guarantees that it is secret and authentic.

```
rule ChanOut_S:
    [ Out_S($A,$B,x) ]
    --[ ChanOut_S($A,$B,x) ]->
    [ !Sec($A,$B,x) ]
rule ChanIn_S:
    [ !Sec($A,$B,x) ]
    --[ ChanIn_S($A,$B,x) ]->
    [ In_S($A,$B,x) ]
```

Chapter 3

Remote attestation protocols

The attestation protocols for automatic establishment of trust through the verification of the integrity of the fog nodes, can be of two types:

- Attestation by Proof: it is a local attestation, which preserves privacy;
- Attestation by Quote: it is a remote attestation, which involves the TPM quote, a structure containing a nonce and the PCRs [2].

Symbols and Abbreviations	**Translation**
Orc	Orchestrator
TPM	Trusted Platform Module
VF	Virtual Function
EK	Endorsement Key
SK	Storage Key
AK	Attestation Key
Vrf	Verifier
Prv	Prover
Trce(r)	Retrieve the binary contents of the object r
h	Hash digest
hmac	Hash-based Message Authentication Code
Vf	Verification
$hk_{(a,b)}$	Symmetric hash key between a and b
Sign(a, b)	Computes the signature over a with b
Sig_a^b	Signature over a with b
proof (a)	TPM's secret value associated to a's hierarchy
TPL(a)	Template for a (attributes)
name(a)	Digest over the public part of a
CC	TPM command code
PCR	Platform Configuration Register
NVPCR	Non-Volatile PCR
mPCR	Mock PCR
mNVPCR	Mock NVPCR

Table 3.1: Table of abbreviations.

In this chapter are presented the protocols for remote attestation that are modeled and evaluated in the next chapters: Attestation by quote and Oblivious remote attestation, which is an update version of the Attestation by Proof protocol. These two protocols exploit TPMs' capabilities, which authenticate the evidence of the integrity state of the PCRs, inferring the correctness of the component.

3.1 Attestation by quote

The attestation has to be executed together with the update of measurements protocol in order to verify the integrity of the device, because policy and configuration can always change and be updated. For this reason, the state of the platform has to be attested periodically, for the node to be considered trusted. Each time there is a change in the policies or there are configuration updates, the state has to be verified, comparing it to the reference hashed digest computed by the Orc. This verification process can be done with the Attestation by Quote protocol.

3.1.1 Update protocol

The Orchestrator, each time there is an update in the configuration, computes the new state, accumulating values in the virtual PCRs (vPCRs), that will be compared to the real ones for attestation. Then, the Orc sends the update request and the Tracer, which given an object identifier returns securely the corresponding binary data, measure the requested file and the TPM extends its PCRs.

The Update protocol is shown in the figure below (Figure 3.1).

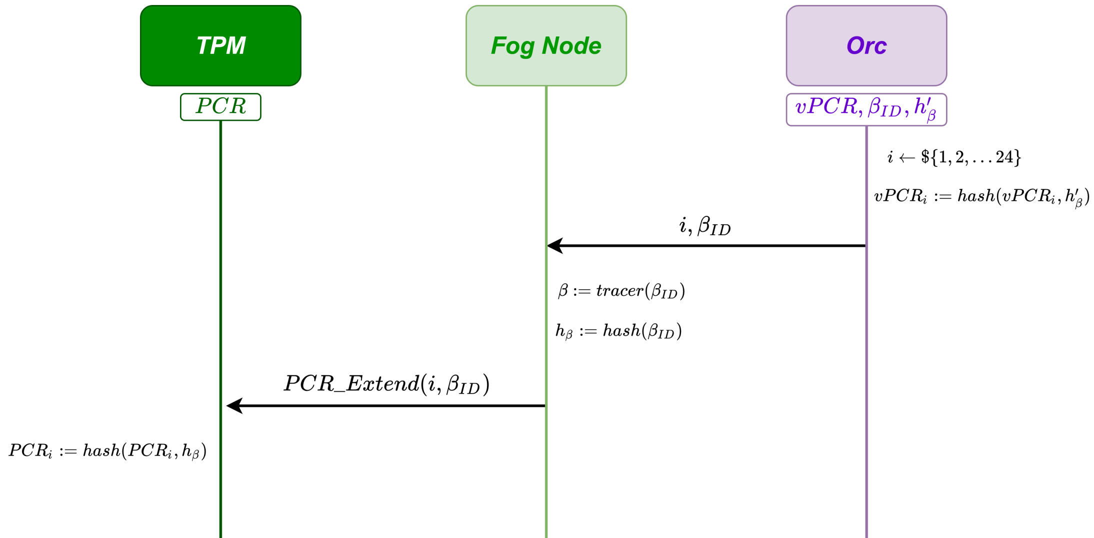

Figure 3.1: Update Measurement protocol.

3.1.2 Attestation by quote protocol

The Attestation by quote protocol is described in the figure below (Figure 3.2). To check the updated PCRs and the current state of the platform, after the update, the Orchestrator sends an Attestation by Quote request, specifying the PCRs to attest (I) and a nonce. The TPM computes its state, an hash of the specified PCRs, and a

Quote containing the computed hash, a nonce and I. Inside the TPM is embedded an RSA key, called Endorsement Key, which is used, in this protocol, to sign the Quote, so as to prove that the Quote was securely built inside it. After the signing operation, the TPM sends the signed Quote to the Orchestrator for verification, that will compare the hashed PCRs with the virtual ones computed by itself.

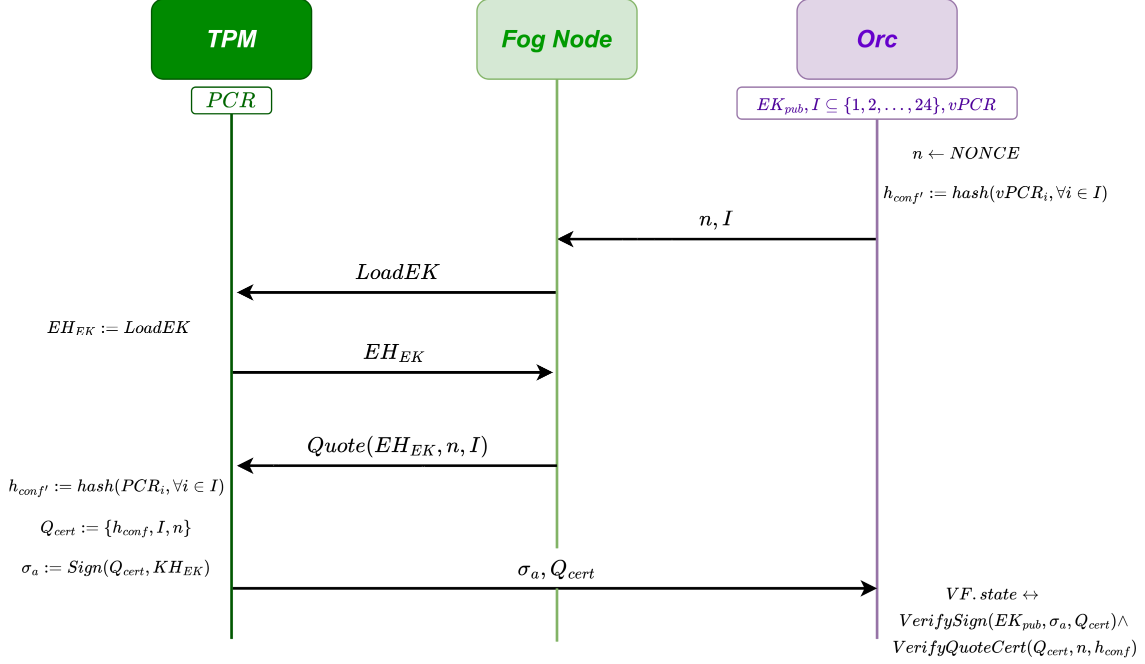

Figure 3.2: Attestation by Quote protocol.

3.2 The Oblivious Remote Attestation

The Oblivious Remote Attestation[3] is a process of attestation in a redefined system, that takes into account a virtualized system, with a virtualized network and an Orchestrator that manages VFs (Virtual Functions) associated with a software TPM as trust anchor.

A possible problem in this redefined system is the exclusive use of the normal registers, the PCRs already existing in the TPM, which are few. This issue limits the number of VFs, because each VF needs a static PCR for measuring and storing its state. In order to solve this problem, NVPCRs have been introduced. NVPCRs are registers created from the Non-Volatile memory: this solution solves the issue, because in theory there can be an unbounded number of NVPCRs. This new types of registers, which imitate the normal PCRs, have to be managed, from creation to deletion: this is the reason why two other protocols of remote administration have been introduced. NVPCRs, together with the static PCRs, are used in all the protocols that are going to be presented.

The Oblivious Remote Attestation (ORA) process allows VFs to check if other VFs can be considered trusted (Figure 3.3). This attestation process starts with the Orchestrator sending a request for the creation of an Attestation Key (AK) to a VF (X). X creates in a secure way the AK using its TPM; after the AK is verified by the Orchestrator, it can be used to sign the challenges that other VFs send to X to check

if it's in a correct state. This protocol does not need exchange of state information, as opposed to the Attestation by Quote, for this reason privacy can be considered maintained during its execution. Another protocol useful in these scenario is the Measurement Update one, which is the protocol managing a configuration update and attesting whether the update was done correctly or the platform is corrupted.

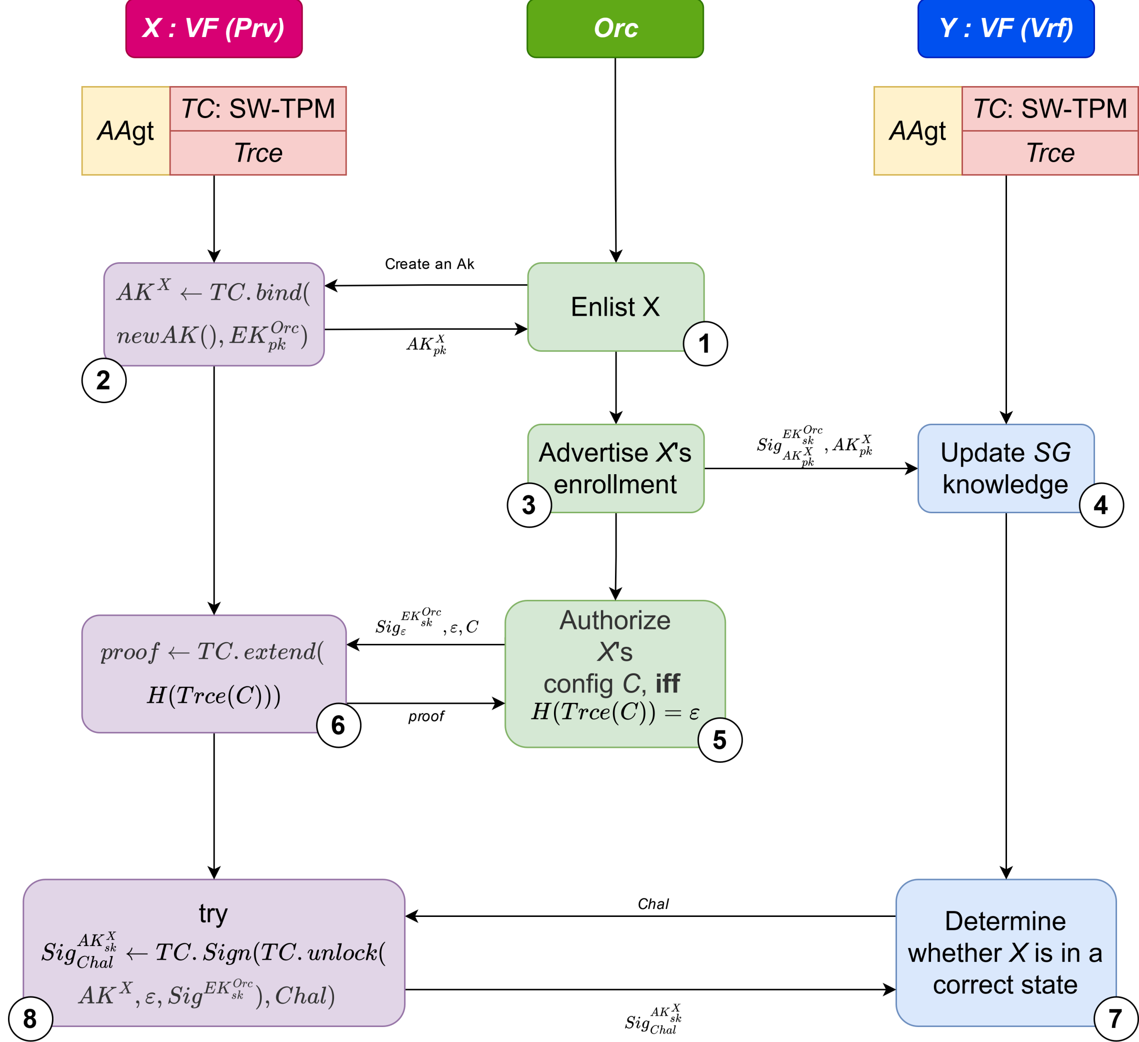

Figure 3.3: Oblivious-based inter-VF Remote Attestation.

3.2.1 Attestation Key creation

The AK creation protocol starts when the Orchestrator sends a message containing the TPL, that is a template of the properties that the AK should have, and an hash (h_{pol}) defined as:

$$h_{pol} = H(H(0...0||CC||name(EK^O)))$$

The parameters of the previous formula are:

- *0...0*: padding value;
- CC: command code;
- $name(EK^O)$: the hash of the public parts of the Orchestrator's Endorsement Key.

X receives this message and sends it to the TPM, together with the Storage Key handle (H_{sk}). The TPM creates:

- Attestation key (AK);
- Attestation key Handle (Hak);
- Ticket: it is the result of an HMAC operation over some internal information, that attests that the Key has been created inside the TPM.

After creating these data, the TPM sends them to X, to receive them back and verifies that the ticket holds, which means that the AK was actually created inside the trusted module. The last step done by the TPM is the creation of the certificate signed with the Endorsement Key. The certificate is sent to the Orchestrator through X and the Orchestrator verifies all the certificate information in order to check if the AK was created legitimately and can be used for future inter-VFs attestations. The command EvictControl is a particular command for the TPM: executing it the TPM stores the AK in order to make it persistent for X's lifetime.

In the Figure 3.4 are described all the steps of the Attestation Key provisioning protocol.

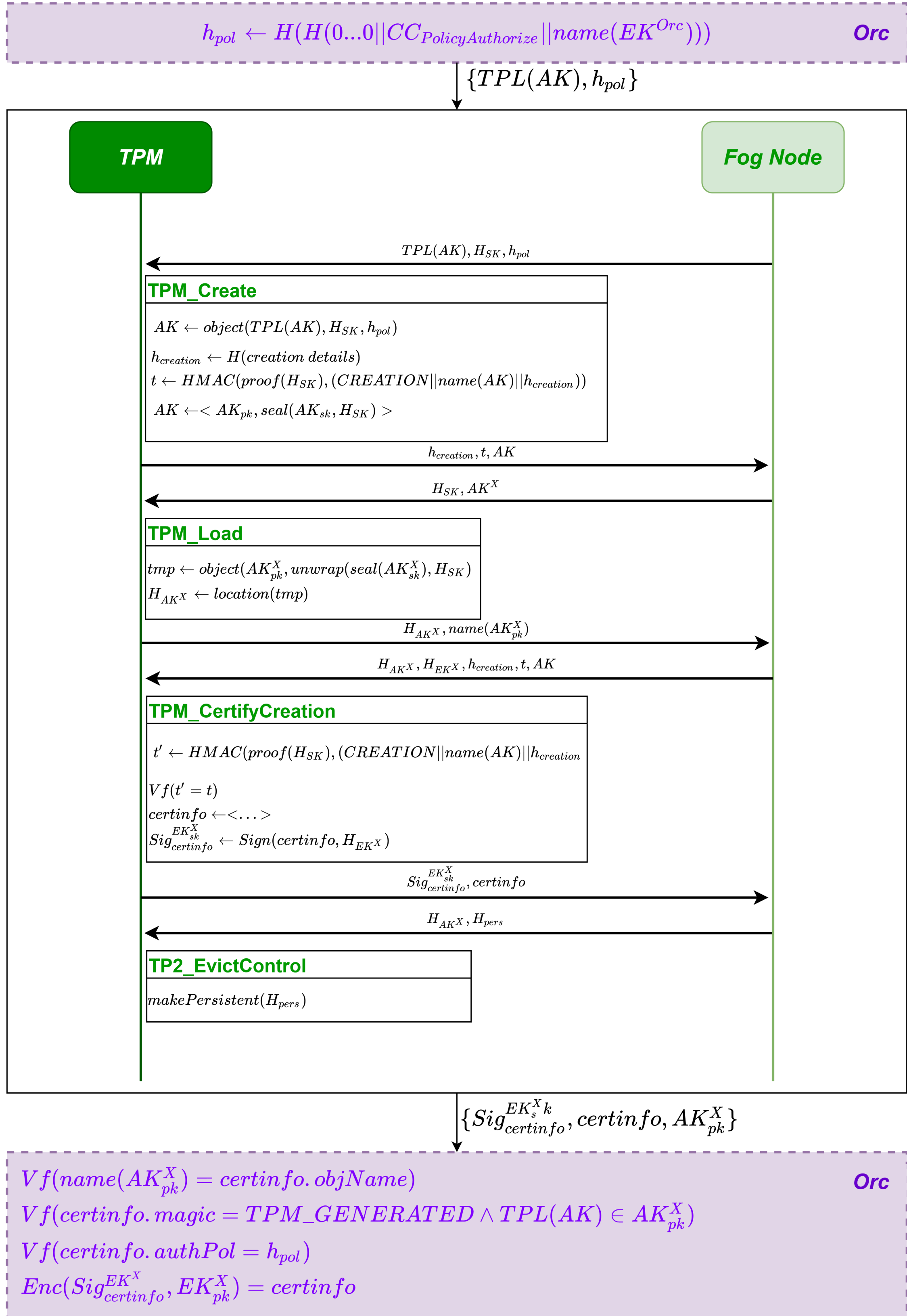

Figure 3.4: Attestation Key creation protocol.

3.2.2 Measurement Update protocol

The Measurement Update protocol is a protocol used to manage configuration updates: the Orchestrator starts the protocol and compares the update done by the TPM on its registers with the one computed by itself in the mock PCRs. The mock PCRs are the registers used by the Orchestrator to emulate the operations and the update done by the TPM and to store the results for future verification. In the context of the Oblivious Remote Attestation, this protocol is important to update measures with the policy authorized by the Orchestrator, after the Attestation Key is created, in order to enable X to prove its configuration correctness using its AK to other VFs. The protocol starts with the Orchestrator measuring locally the new configuration and authorizing the expected policy digest. Then the Orchestrator sends the FQPN, which is the path of the configuration file, together with the update request to X. The update request is a message defined as:

$$Req_{update} =< h_{pol}, H(h_{pol}), Sig_{H(h_{pol})}, idx >$$

The values of the update request are:

- h_{pol} : the authorized policy digest;
- $H(h_{pol})$: the hash of the authorized policy digest;
- $Sig_{H(h_{pol})}$: the signature over the hash of the authorized policy digest computed with the Orchestrator's Endorsement Key;
- idx : the index of the register to update with the hash of the file.

The measure of the file is done with the Trce function. After the measure, X's TPM verifies the policy digest and creates the ticket attesting it has been verified. The TPM then proceeds to extend the specified PCR with the new measure during an HMAC session. At the end of the process the TPM certifies the current session digest and the Orchestrator verifies the digest comparing it with the one it calculates locally. In the Figure 3.5 are described all the steps of the Measurement Update protocol.

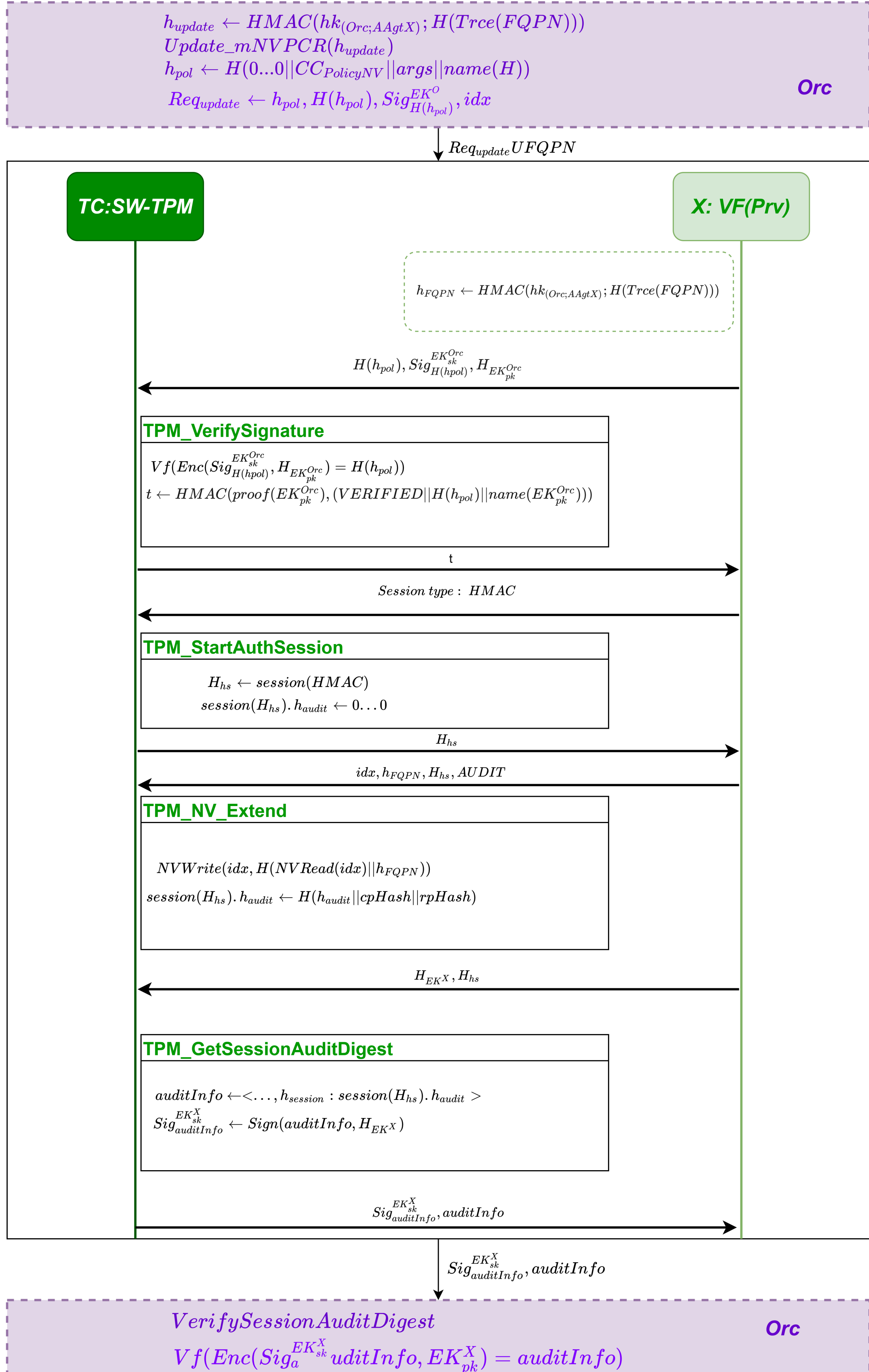

Figure 3.5: Measurement Update protocol.

3.2.3 ORA protocol

The Oblivious Remote Attestation protocol, is a revisited attestation protocol that consists in an updated version of the Attestation by Proof protocol in the redefined system. The ORA protocol is used for remote attestation between two nodes:

- Y: the node that wants to know if the other one is trustworthy;
- X: the node that has to prove its trustworthiness.

if a VF(Y) wants to check the integrity of another VF(X) sends a challenge, a nonce, which has to be signed by X with its Attestation Key, created and verified with the previous protocol (Attestation Key creation protocol 3.2.1). This protocol starts when Y sends a nonce to X. The nonce has to be signed with the Attestation Key, but in order to use the Attestation Key, the values of X's PCRs must be in a specific state, a state authorized by the Orchestrator. X sends to the TPM the authorized policy (P), which contains the values of the PCRs that have been authorized, and the ticket (T), which proves that P is signed by the Orchestrator. Then, the TPM computes its P and T and checks them against the values received by X with the PolicyAuthorize command. If the verification holds, the TPM can sign the nonce with the Attestation Key (Sign command) and send it back through X to Y, which can verify the signature and confirm that X is trusted. In the Figure 3.6 are described all the steps of the ORA protocol.

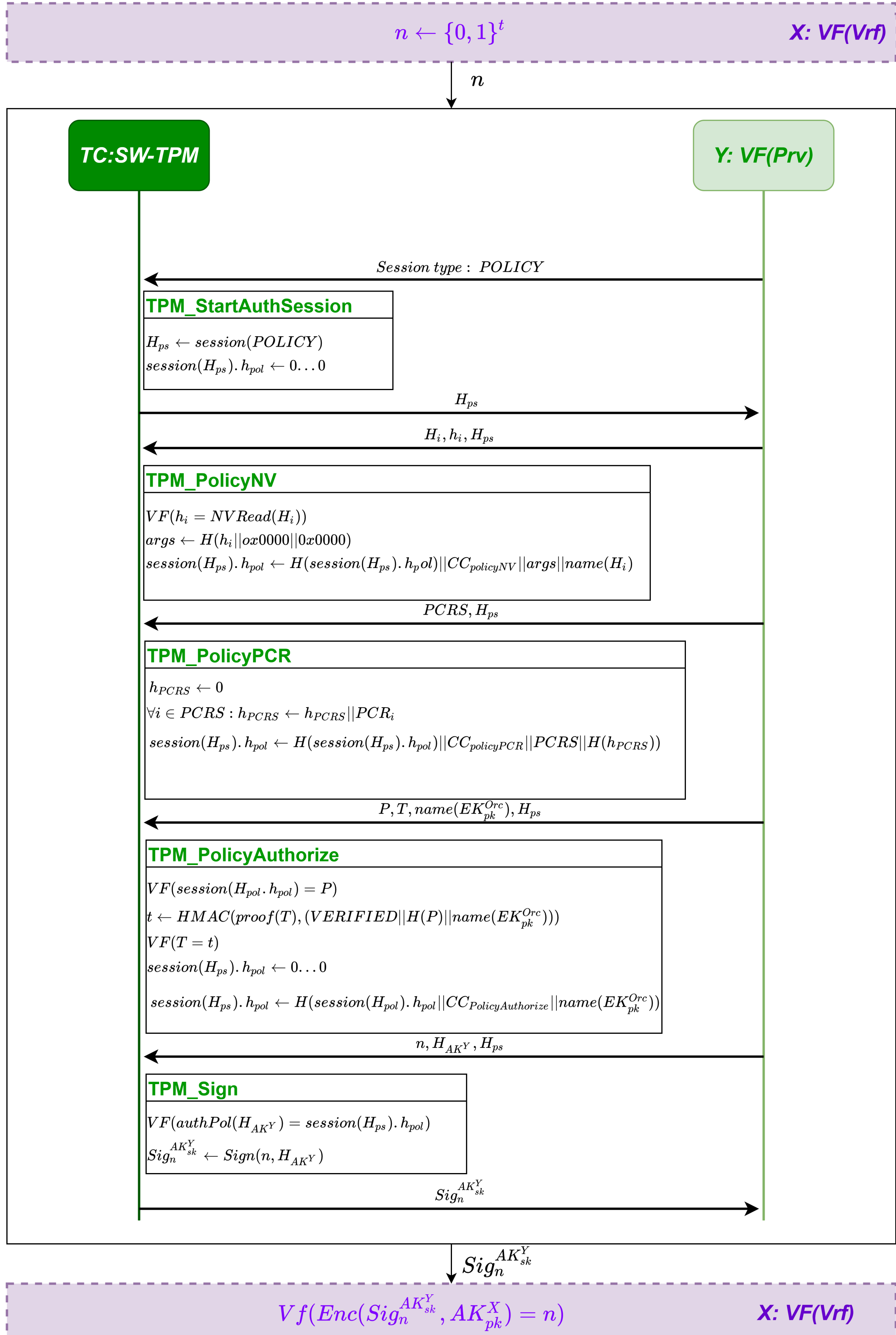

Figure 3.6: Inter-VFs Oblivious Remote Attestation protocol.

3.2.4 NVPCRs administration

As explained at the start of this section, NVPCRs have to be managed and administrated, from creation to deletion, as they can also be reset and recreated during run-time. The two protocols useful to remotely administrate NVPCRs, are called the Attaching protocol and the Detaching protocol. Creation and deletion of these registers have to be authorized by the Orchestrator with a flexible policy.

Attaching NVPCR protocol

For attaching a NVPCR to a specific virtual function X, the Attaching NVPCR protocol can be used. This protocol, shown in the Figure 3.7, starts with the Orchestrator sending a request to X to add a new NVPCR. The add request is defined as:

$$Req_{add} =< idx, TPL(idx), IV, h_{pol} >$$

The values contained in this message are:

- idx: NV identifier;
- TPL(idx): template with the attributes of the NV slot;
- IV: initial value to extend;
- h_{pol}: the authorization policy.

Upon an add request sent by the Orchestrator, the TPM execute three commands:

- NV_DefineSpace: TPM creates the NVPCR;
- NV_Extend: TPM extends the NVPCR with initial value;
- NV_Certify: TPM certifies the NVPCR.

After these three operations, the Orchestrator has to verify that everything holds, that the signature is correct, that the values are the ones expected and the register is bound to the authorization policy it has created.

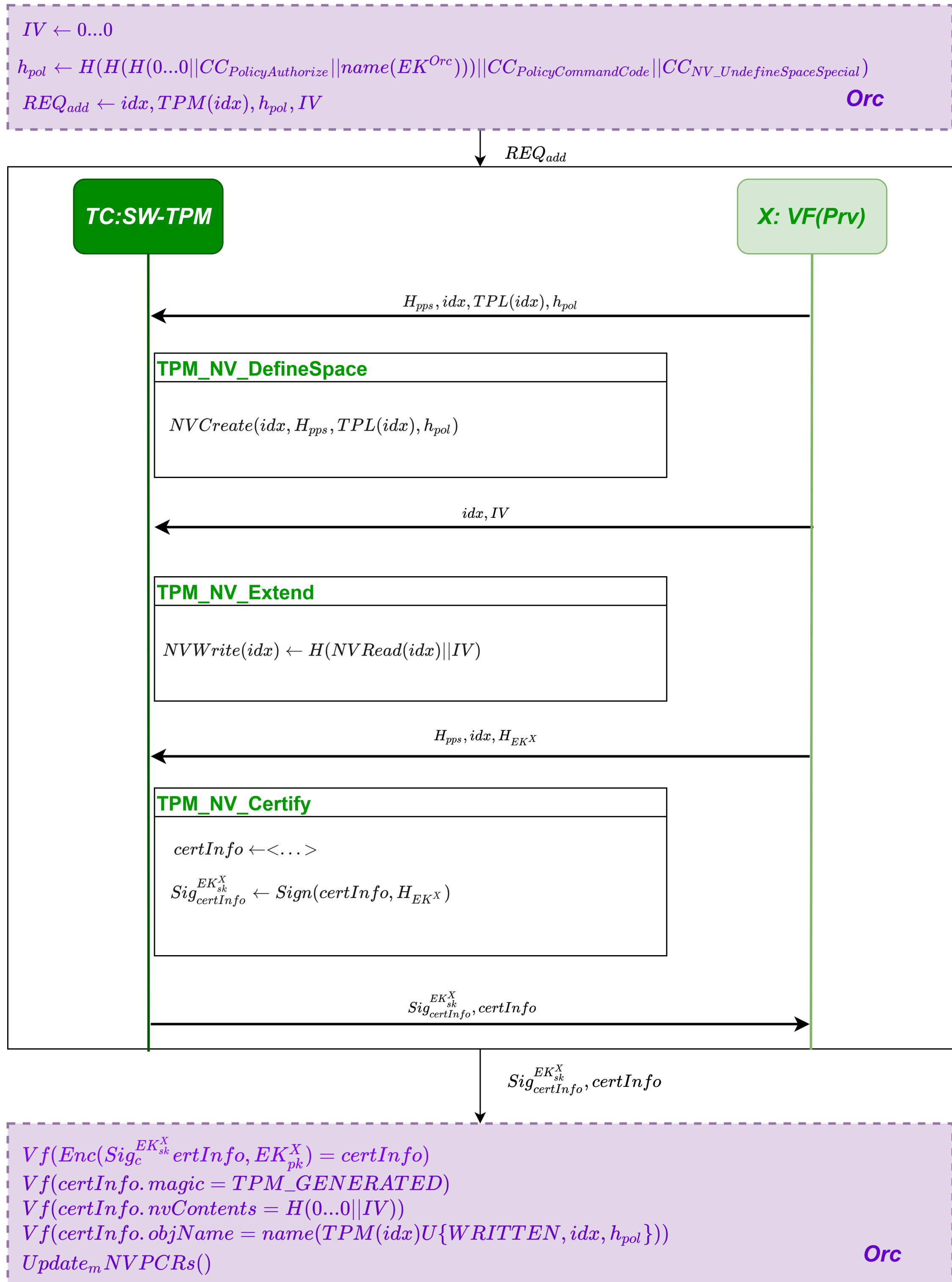

Figure 3.7: Attaching NVPCRs protocol.

Detaching NVPCR protocol

The Figure 3.8 below, shows the details of the Detaching NVPCR protocol. This protocol starts with a fresh session, with X sending a nonce to the Orchestrator.

Then, the Orchestrator sends a delete request. The delete request is defined as:

$$Req_{del} =< idx, h_{cp}, Sig_{aHash}, h_{pol}, H(h_{pol}), Sig_{H(h_{pol})} >$$

The values contained in the delete request are:

- idx: NVPCR's identifier;
- h_{cp}: CP digest, digest over the command parameter, used to restrict the policy only to the correct register;
- Sig_{aHash}: signature over the aHash value (hash of the nonce and the h_{cp} value) with the Orchestrator's Endorsement Key;
- h_{pol}: authorized policy;
- $H(h_{pol})$: digest over the authorized policy;
- $Sig_{H(h_{pol})}$: signature over the hashed policy digest value with the Orchestrator's Endorsement Key.

The TPM verifies the signatures of the h_{pol} value and executes the command PolicySigned with the aHash value, signed by the Orchestrator: this command updates the session's policy digest to specify that the aHash was signed by the Orchestrator. The following commands executed are the PolicyAuthorize one, which checks the value of h_{pol} and specifies that the policy authorized is correct, and the PolicyCommandCode, which restricts the session to the register in question. Last command executed is NV_UndefineSpaceSpecial, that is the one that actually deletes the NVPCR.

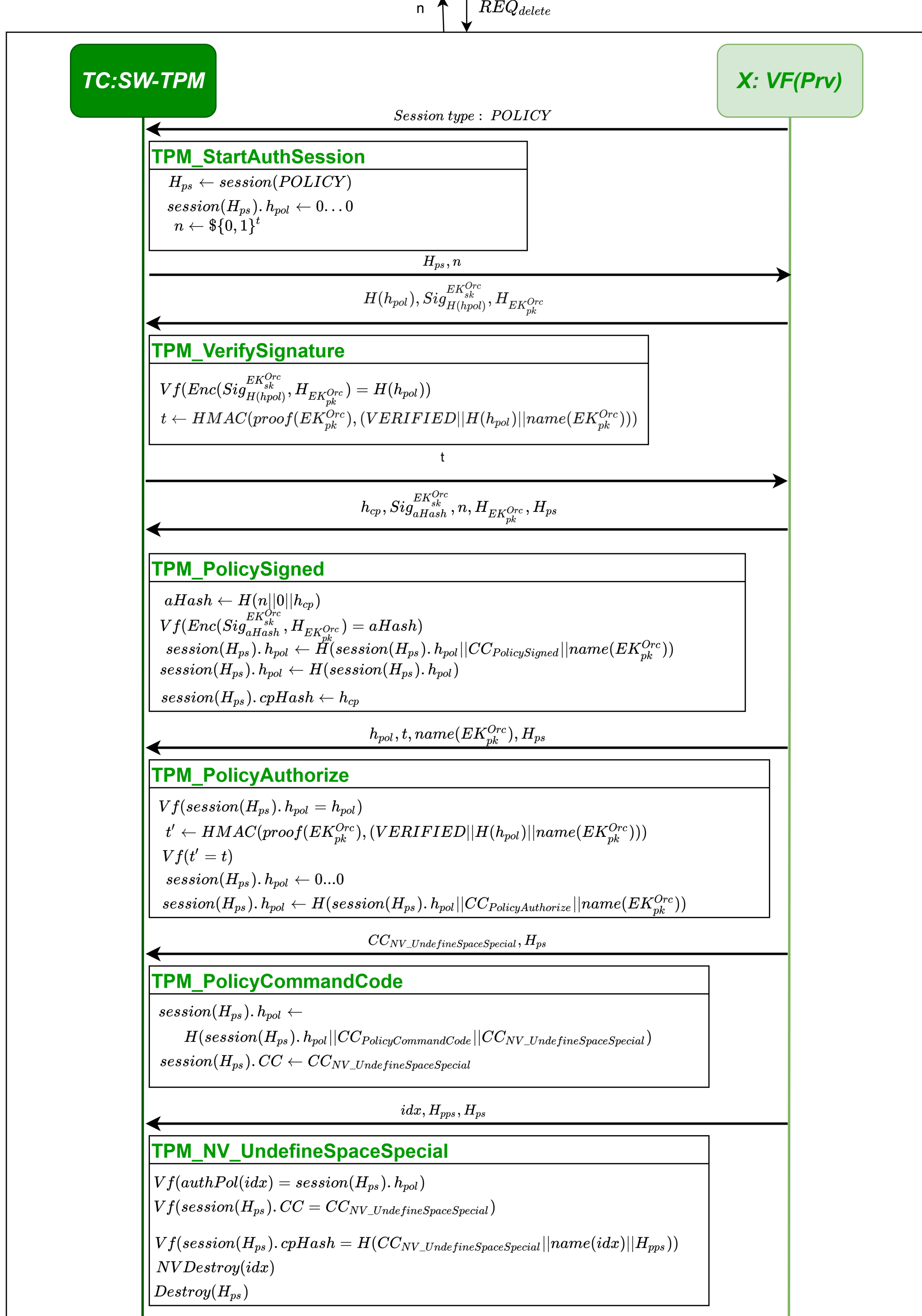

Figure 3.8: Detaching NVPCRs protocol.

Chapter 4

Objective of the thesis

This thesis work is done within the context of the European project called Rainbow, which aims to provide a trusted fog computing platform for managing heterogeneous IoT (Internet of things) devices. In this chapter the objective of the thesis is presented, explaining how it was achieved.

4.1 Objective of the Thesis

In a fog computing environment there can be a great number of security issues, in particular, it is important to provide an integrity verification mechanism. Rainbow designed the required security protocols to provide remote attestation in the platform, which is the process of attesting if a device can be considered trusted, verifying its integrity and its operational correctness. The Orchestrator is the one entity that has to verify if the state of the device is correct, in order to know if it can be considered trusted. The objective of this thesis is to verify the correctness of the designed protocols for remote attestation, which have to guarantee a certain level of security. Rainbow has defined specific properties, from which axioms of trust can be extracted, that these protocols have to satisfy. Trust has to be guaranteed by all the entities involved in the attestation process: Fog Nodes, TPMs and Orchestrator [1]. These defined properties include among the others:

- VF configuration correctness: device's configurations must adhere to the policies of attestation authorized by the Orchestrator;
- Attestation Key protection: the Attestation Key has to be created securely and kept secret to the adversary, so that the unforgeability is ensured;
- Immutability: the process of measurement must not change, the tracer always measures correctly;
- Liveness and controlled invocation: attestation requests must reach the device and the agent has to respond in a limited time;
- VF operational correctness: the Orchestrator has to always be able to verify the correctness and the control flow safety of a device;
- SGC (Service Graph Chain) operational correctness: the Orchestrator has to always be able to verify the trustworthiness and the workflow safety of the SGC.

The security of a protocol can be evaluated using different techniques. In this thesis, the correctness of remote attestation protocols is analysed using the Formal Verification, which is an a-priori security verification of a mathematical model. This work of thesis has the purpose to analyse the designed protocols for remote attestation using a tool called Tamarin Prover. In order to properly use the Tamarin Prover, the protocols have been symbolically modelled, with the specific syntax required by Tamarin. In particular, the modelled protocols are:

- Attestation by quote protocol: protocol of remote attestation based on the Quote, a structure containing the registers value (Section 3.1);
- Attestation Key creation protocol: protocol for the creation of the Attestation Key bound to a node (Section 3.2.1);
- Measurement Update protocol: protocol for managing the update of the configurations (Section 3.2.2);
- Oblivious Remote Attestation protocol: protocol of remote attestation between two nodes (Section 3.2.3);
- Attaching NVPCRs protocol: protocol for creating a NVPCR (Section 3.2.4);
- Detaching NVPCRs protocol: protocol for deleting a NVPCR (Section 3.2.4).

The symbolic models of these protocols have been given to the Tamarin Prover as input, comprehensive of some lemmas to verify, which describe the properties that the protocols have to fulfill.

Tamarin Prover, given these inputs, automatically proves the correctness of the protocols, in an unbounded number of sessions and in presence of an adversary. The results of the verification can be seen from the GUI of the framework, that provides visual representation of the execution, showing graphs from which it was possible to analyse the flow of the protocols and all the possible sources for each fact. Using these graph it has been analysed the correctness of the models and the security of the protocols, highlighting, in particular, the possible problems and the reasons behind those problems.

Chapter 5

Design

This chapter describes how the protocols for attestation have been symbolically modeled, in order to understand, using the Tamarin prover, their level of security in presence of a DY adversary.

5.1 Attestation by quote

The Attestation by quote protocol and the update protocol are modeled together, because each time there is an update, an attestation must be executed to verify the integrity of the device. These two protocols are modeled using a public channel between the fog node and its TPM. The model is comprehensive of thirteen rules, which describe the states of the execution. The entities involved in the Attestation by quote are four:

- O: the Orchestrator;
- F: the Fog Node;
- TPM: the trusted component;
- T: the Tracer.

In the model the following built-ins (set of equations) are used:

- Hashing: it is used for the hash operations, for example the hash of the configuration;
- Signing: it is used for signing operations, in this model for signing the quote.

Before the actual protocols rules, some rules to initialize the entities are needed. The Orchestrator is created in the Orc_init rule, the Tracer in the Traces_init rule and the platform (Fog Node and TPM) in the Platform_setup rule. The Platform_setup rule pairs up the fog node and its TPM. It also loads the Endorsement Key of the TPM (ek) through the ek_get function, that accepts the Endorsement key handle as parameter. This key is used to sign the quote, therefor it is stored in the !Ek($TPM, ek) persistent fact. This rule has two important restrictions:

- OnlyOnce: it is a restriction used to specify that the rule has to be executed only one time;

- Unique_Pairing: it is a restriction used to specify that the pair of F and TPM is unique.

```
rule Platform_setup:
let
    ek = ek_get(handle_ek_get)
in
[
]
--[
    Platform_setup($TPM, $F)
    , PlatformInit()
    , OnlyOnce()
    , Create($TPM)
    , Create($F)
    , Unique_Pairing($F), Unique_Pairing($TPM)
 ]->
[
    St_PlatformInit($TPM, $F)
    , !Ek($TPM, ek)
]
```

Another important rule that is used in the model is the TPM_EKReveal rule, which models the dynamic compromise of the Endorsement Key: it reads the EK and sends it in the public channel for the adversary to take.

```
rule TPM_EKReveal:
[ !Ek($TPM, ek) ] --[ Reveal_ek($TPM) ]-> [ Out(ek) ]
```

5.1.1 Update protocol model

The model of the update protocol, described in the Section 3.1.1, starts with the Orc_send_update_req, which starts from an initial state where all the entities are initialized. The Orchestrator gets the virtual PCRs values and the file that have to extend the PCRs by means of private functions, defines as follows:

- index_get/0 [private]: function without any input;
- id_get/0 [private]: function without any input;
- file_get/1 [private]: function with one input value, which is the id of the file;
- vpcr_I_get/1 [private]: function with one input value, which is the index of the register to be updated.

The rule ends with the Out fact, containing the file id and the index of the PCRs, and two other persistent fact used to store the public part of the Endorsement Key and the virtual registers updated with the index for future verification.

```
rule Orc_send_update_req:
let
    index = index_get()
    id = id_get()
    vpcr_i = vpcr_I_get(index)
    file = file_get(id)
    h_beta = h(file)
    vpcr_i = h(<vpcr_i, h_beta>)
    ek_pub = pk(ek_get(handle_ek_get()))
in
[
    St_tracer($T)
    , St_Orc($O)
    , St_PlatformInit( $TPM, $F)
]
--[
    Orc_send_update_req()
    , Running($O, $T, <'update_req_to_tracer', index>)]->
[
    Out(<$O, id, index>),
    !St_Orc_vPCR_i($O, vpcr_i, index),
    !St_O_ek_pk($O, $F, ek_pub)
]
```

After the update request, another agent interacts with the platform and the Orchestrator: the Tracer. Actions done by the tracer, in particular the hash of the file, are modeled in the Tracer_update rule. The hash sent by the Tracer is then forwarded by F to the TPM, together with the index of the register to be updated, in the F_update_measure rule.

The last rule used to model the update phase, is the TPM_update_measure one. This rule models how the TPM, starting from an initial state with its PCRs and the hashed values received from the fog node, extends the PCRs and transitions to a state where the new configuration is stored in the !St_TPM_pcr_i($TPM, pcrs_i, index) fact.

```
rule TPM_update_measure:
let
    pcrs_i = pcrs_I_get(index)
    pcrs_i = h(<pcrs_i, hb>)
in
[
    In(<$F, hb, index>)
]
--[
    TPM_update_measure(), UniqueExecUpdateM('TPM_updates_PCRi')
    , Commit($TPM, $F, <'update_req_to_TPM', index>)
]->
```

```
[
    !St_TPM_pcr_i($TPM, pcrs_i, index)
]
```

5.1.2 Attestation by quote protocol model

The attestation phase starts with the O_send_nonce rule, which models the Orchestrator producing a nonce (n) in its initial state with the Fr fact and computing its configuration hash. Agent O then sends the nonce to the Fog Node with the index of the PCRs to attest (i).

```
rule O_send_nonce:
let
    I = I_get()
    nonce = ~n
    h_conf_Orc = h(updated_vpcrs_get(vpcr_i))
in
[
    Fr(~n)
    , !St_Orc_vPCR_i($O, vpcr_i, index)
]
--[
    O_send_nonce()
    , Send_nonce_to_FN(~n)
    , Running($O, $F, <'init_attestation', nonce>)
  ]->
[
    Out(<$O, nonce, I>)
    , St_Orc_nonce_I($O, $F, nonce, I)
    , St_O_hconf($O, h_conf_Orc)
]
```

The values sent by the Orchestrator are forwarded to the TPM by the Fog Node (F_send_nonce_to_TPM rule) and then taken by the TPM from the channel using an In fact. The TPM constructs the quote, signs it and sends it back to be verified: this is modeled in the TPM_signAndSend rule. Again, the Fog Node receives the massange from the TPM and forwards it to the Orc (FogNode_sendSigned rule).

```
rule TPM_signAndSend:
let
    h_conf = h(updated_pcrs_get(pcrs_i))
    QCert = <h_conf, I, nonce>
    QCert_sig = sign(QCert, ek)
in
[
    In(<$F, nonce, I>)
```

```
    , !St_TPM_pcr_i($TPM, pcrs_i, index)
    , !Ek($TPM, ek)
]
--[
    TPM_signAndSend()
    , Send_sig($TPM, QCert)
    , Commit($TPM, $F, <'init_attestation_to_TPM', nonce>)
    , Running($TPM, $F, <'send_signed_to_F', QCert>)
    , UniqueExecSign('TPM_signing')
]->
[
    Out(<$TPM, QCert_sig, QCert>)
]
```

The last step of the protocol, is modeled using the Orc_receiveSigned_verify rule: the Orchestrator takes the quote and the signature from the public channel and, knowing the public part of the Endorsement Key, previously stored in the !St_O_ek_pk($O, $F, ek_pub) fact, verifies the signature and the quote.

```
rule Orc_receiveSigned_verify:
let
    orc_QCert = <h_conf_Orc, I, nonce>
in
[
    !St_O_ek_pk($O, $F, ek_pub)
    , St_Orc_nonce_I($O, $F, nonce, I)
    , !St_Orc_vPCR_i($O, vpcr_i, index)
    , In(<$F, QCert_sig, QCert>)
    , St_O_hconf($O, h_conf_Orc)
]
--[
    Orc_receiveSigned_verify()
    , Eq(verify(QCert_sig, QCert, ek_pub),true)
    , Commit($O, $F, <'send_signed_to_O', QCert>)
    , Recv($O, QCert)
    , Honest($O)
    , Honest($F)
    , Honest($TPM)
    , Authentic($TPM, QCert)
    , Eq(orc_QCert, QCert)
]->
[]
```

5.2 Oblivious Remote Attestation

For the models of the Oblivious Remote Attestation process, it has been taken into account the redefined system described in Section 3.2. Before describing the model of the Oblivious Remote Attestation protocol, the model of the Attestation Key creation protocol and the Measurement update one will be presented.

The Attestation key creation protocol creates and provides a VF with its Attestation Key, that is used in the ORA protocol for signing a challenge for attestation.

The Measurement update protocol is the protocol used for configuration updates: having a path name of a configuration file and the registers, the Orchestrator locally measures and authenticates the configuration measurement and authorizes the expected policy digest, and the TPM extends its registers, both PCRs and NVPCRs.

At the end of this chapter, also the models of the two protocols managing NVPCRs will be presented: Attaching and Detatching protocols.

5.2.1 Attestation Key creation protocol model

The entities involved in the AK creation are three:

- O: Orchestrator;
- X: Virtual Function;
- TPM: trusted component (software).

The states of the execution are described by twelve rules and the interactions between the entities are modeled with a public channel.

In the model the following built-ins (set of equations) are used:

- Asymmetric-encryption: used for the seal operation;
- Hashing: it is used for the hash operations, for example the hash of the configuration;
- Signing: it is used for signing operations, in this model for signing the quote.

In the execution of the Attestation Key creation protocol there are in particular two keys that do not have to be compromised: the storage key, used for the seal operation, and X's endorsement private key, used for the signature of certificate information. Their reveal is modeled with two specific rules: TPM_SK_reveal and X_ek_priv_get_reveal.

The first rule is the Platform_setup rule: it sets up X and the TPM, creating some persistent facts containing the handles (for which values are retrieved from specific private functions), that can be consumed more than one time over the execution in order to get the needed keys:

- hsk: used to get the Storage Key;
- hek_x: used to get the private part of the Endorsement key of X.

The restriction used in this rule, as well as the Equality restriction, are the ones already used in the Attestation By Quote model (Section 5.1).

```
rule Platform_setup:
let
    hsk = hsk_get()
    hek_x = hek_X_get()
```

```
    creation_details = creation_details_get()
    storage_key = sk_get(hsk)
    ek_x_priv = ek_x_priv_get(hek_x)
in
[
]
--[
    Platform_setup($TPM, $X)
    , PlatformInit()
    , OnlyOnce()
    , OnlyOnceP('Platform')
    , Create($TPM)
    , Create($X)
    , Unique_Pairing($X), Unique_Pairing($TPM)
 ]->
[
    !St_key_handles_X($X, hsk, hek_x)
    , St_creation_details_TPM($TPM, creation_details)
    , St_PlatformInit($TPM, $X)
    , !Sk_TPM($TPM, storage_key)
    , !Ekx_TPM($TPM, ek_x_priv)
]
```

The other initialization rule used is the Orc_init rule, which retrieves the handle of the Endorsement Key(EK) of the Orchestrator and its name, consisting in an hash of the public part of its EK, through the name_Orc function. After the initializations, the Orchestrator sends h_pol and the template containing the attributes needed for the AK creation to X (Orc_send rule), that forwards them (X_send_hpol rule) to its TPM together with the handle of the Storage Key (SK). Subsequently, following the flow of the protocol, the creation of the Attestation key is modeled in the TPM_create_AK rule. The rule TPM_create_AK creates the ak object, the AK and the ticket(using the function hmac/2). The AK is a pair of keys that contains the public key and the sealed private key. The secret part of the AK key is retrieved from the function ak_private_get/1, which takes as input the AK object previously created with the obj/3 function. The seal operation is described in Section 2.1.1: it is an encryption done using the Storage Key of the TPM. This rule also describes the transition to an Out fact, by which the TPM sends the ticket, the AK and some creation details to X, that sends the message back and stores the ticket in the St_handles_t_hcreate_X_3 linear fact (X_send_akx rule).

```
rule TPM_create_AK:
let
    ak = obj(template_ak, hsk_x, h_pol)
    h_creation = h(creation_details)
    ak_sk = ak_private_get(ak)
    ak_pk = pk(ak_sk)
    ak_name = h(ak_pk)
```

```
    t = hmac(proof_hsk_get(), <'CREATION', ak_name, h_creation>)
    sealed_ak_sk = aenc(ak_sk, storage_key)
in
[
    St_creation_details_TPM($TPM, creation_details)
    , !Sk_TPM($TPM, storage_key)
    , In(<$X, h_pol, template_ak, hsk_x>)
]
--[
    TPM_send_AK_t()
    , Honest($TPM) //
    , Commit($TPM, $X, <'init_hpol', h_pol>)
    , Running($TPM, $X, <'create_ak', ak>)]->
[
    St_ak_name($TPM, ak_name)
    , St_h_pol($TPM, h_pol)
    , St_ak_pub($TPM, ak_pk)
    , Out(<$TPM, h_creation, t, <ak_pk, sealed_ak_sk>>)
]
```

After the creation of the AK's handle (TPM_load rule) using the function location/1, the ticket has to be sent back to the TPM from X for verification: if the ticket is correct, the TPM creates the certificate for the generated AK and signs it with the Endorsement key. This actions are modeled in the TPM_CertifyCreation rule, that produces an Out fact of the signature and the certificate. The verification of both the signature and the certificate fields that has to be done by the Orchestrator is modeled in the Orc_verify rule: in particular, it is modeled using the Equality restriction to verify the signature and compare the fields of the certificate with the correct values.

```
rule TPM_CertifyCreation:
rule TPM_CertifyCreation:
let
    t_primo = hmac(proof_hsk_get(), <'CREATION', ak_name, h_creation>)
    certInfo_objName = ak_name
    certInfo_magic = 'TPM_GENERATED'
    certInfo_authPol = h_pol
    signature = sign(<certInfo_authPol, certInfo_magic,
                        certInfo_objName>, ek_x_priv_get(hek_x))
in
[
    In(<$X, hak_x, hek_x, h_creation, t>)
    , !Sk_TPM($TPM, storage_key)
    , St_handle_ak_x($TPM, ak_name, hak_x)
    , St_h_pol($TPM, h_pol)
    , St_ak_pub($TPM, ak_pk)
]
```

```
--[
    TPM_verify_ticket(), Eq(t_primo, t)
    , UniqueExecSign('TPM_signing')
    , Secret(ek_x_priv_get(hek_x))
    , Send_signed_to_Orc($TPM, <certInfo_authPol,
                                certInfo_magic, certInfo_objName>)
    , Commit($TPM, $X, <'send_ticket', t>)
    , Running($TPM, $O, <'send_signed', signature>)
  ]->
[
    St_certInfo(certInfo_objName, certInfo_magic, certInfo_authPol)
    , !Attestation_key_persistent($TPM, hak_x)
    , Out(<$TPM, signature, <certInfo_authPol, certInfo_magic
                , certInfo_objName>, ak_pk >)
]



rule Orc_verify:
[
    In(<$TPM, certSigned,
            <certInfo_authPol, certInfo_magic, certInfo_objName>
            , ak_pk>)
    , St_Orc_hpol($O, h_pol, template_ak)
    , !St_key_handles_X($X, hsk, hek_x)
]
--[
    Orc_verify()
    , Authentic($TPM,<certInfo_authPol, certInfo_magic
                        , certInfo_objName>)
    , Honest($O)
    , Honest($TPM)
    , Recv_signed($TPM, <certInfo_authPol, certInfo_magic
                        , certInfo_objName>)
    , Commit($O, $TPM, <'send_signed', certSigned>)
    , Eq(h(ak_pk), certInfo_objName)
    , Eq(certInfo_magic, 'TPM_GENERATED')
    , Eq(certInfo_authPol, h_pol)
    , Eq(verify(certSigned,
                <certInfo_authPol, certInfo_magic
                        , certInfo_objName>,
                   ek_x_pub_get(hek_x)), true)
]->
[
]
```

5.2.2 Measurement update protocol

The agents interacting in the ORA protocol are:

- X: VF;
- TPM: X's trusted component (software);
- O: Orchestrator;
- T: Tracer.

In order to model this protocol is taken into account only the case having the Boolean value equal to True, which means that the registers to be extended are the NVPCRs.

The Measurement update protocol has been modeled by means of twenty-one rules. Two of these rules are the ones that model agents keys used for signing operations being compromised: rule Ek_sk_X_reveal and rule Ek_sk_O_reveal.

The creation and initialization of the entities involved are done by three rules: the Platform_setup rule, which stores in persistent facts the handles of the keys and X's EK; the Orc_init rule, which stores in persistent facts the specific configuration path and the PCR's id; the Tracer_init rule. The tracer is used to model the Trce function, that, given the configuration path (fqpn value) returns the configuration. This function is used by both the Orchestrator and X. The Orchestrator uses it to construct the update request (rule Orc_update_request), which has the h_update value as one of its fields, that is an hmac of the hash of the configuration sent by the Tracer with the key shared between the Orchestrator itself and the virtual function, needed for the configuration measure authentication. X uses the Tracer to compute the same hmac value that will be used to extend the PCRs.

```
rule Orc_update_request:
let
    h_update = hmac(hk_O_X, h(trce_fqpn))
    h1 =  h(<h_update>)
    args = h(<h1, '0', '0'>) //h1?
    hpol = h(<'00000000', 'CCpolicynv', args, name_Orc>)
    h_hpol = h(hpol)
    h_hpol_signature = sign(h_hpol, ek_sk_O)
    req_update = <hpol, h_hpol, h_hpol_signature, idx>
in
[
    !St_trce_fqpn_O($O, trce_fqpn)
    , !St_fqpn_O($O, fqpn)
    , !St_hk_O_X($O, $X, hk_O_X)
    , !St_idx_O($O, idx)
    , !Ek_sk_O($O, ek_sk_O)
]
--[
    Orc_update_request()
    , Running($O, $X, <'O_sends_update_request', req_update>)
```

```
    , Orc_send_updateReq_with_signature($O, h_hpol)]->
[
    Out(<$O, fqpn, req_update>)
]
```

After the tracing phase and after taking from the public channel the update request created by the Orchestrator, X sends the hash of the hash of the policy, its signature, done by the Orchestrator, and the handle of Orc's EK to its TPM. The TPM verifies the signature using the public part of Orc's EK and computes the ticket value, an hmac of a value that specifies that the signature has been verified and sends it to X with an Out fact.

```
rule TPM_VerifySignature:
let
    ticket = hmac(proof_get(), <'VERIFIED', h_hpol, name_Orc()>)
in
[
    In(<$X, h_hpol, h_hpol_signature, hek_O_pk>)
]
--[
    TPM_VerifySignature()
    , Commit($TPM, $X, <'X_sends_to_TPM', h_hpol >)
    , Running($TPM, $X, <'TPM_sends_ticket', ticket >)
    , Eq(verify(h_hpol_signature,h_hpol,ek_pk_O_get(hek_O_pk)),true)
    , Honest($TPM)
    , Honest($X)
    , Honest($O)
    , Authentic_Orc_msg($O, h_hpol)
]->
[
    Out(<$TPM, ticket>)
]
```

The TPM_StartAuthSession command is modeled in a rule with the same name, which produces an Out fact containing the handle of the session. After this command, the registers have to be extended. In order to extend the NVPCRs, X sends the hmac, computed before over the hash of the configuration value retrieved by the Tracer, and the id of the registers. TPM receives X's message through an In fact and extends the specified NVPCRs and construct the hash of the session (h_audit): this is modeled in the TPM_NV_Extend rule. In order to read the NVPCRs and extend them, two function have been declared: nvRead/1 and nvWrite/2, both private.

```
rule TPM_NV_Extend: //nv true
let
    nv_write = nvWrite(idx, h(<nvRead(idx), h_fqpn>))
```

```
    h_audit = h(<h_audit_init, cpHash_get(), rpHash_get()>)
in
[
    In(<$X, idx, h_fqpn, handle_hs, audit_msg>)
    , Session($X, $TPM, h_hs, h_audit_init)
]
--[
    TPM_NV_Extend()
    , Commit($TPM, $X, <'X_send_h_fqpn_true', h_fqpn>)
]->
[
    Session_2($X, $TPM, h_hs, h_audit)
]
```

The last TPM command is modeled in the TPM_getSessionAuditDigest rule, in which the TPM signs information of the session (h_audit) with X's EK. X forwards the signature to the Orchestrator and the Orchestrator verifies the signature and the the hash of the session (h_audit) checking it against the one computed by itself using the correct expected values.

```
rule TPM_getSessionAuditDigest:
let
    auditInfo_hsession = h_audit
    auditInfo_signature = sign(auditInfo_hsession, ek_sk_X)
in
[
    In(<$X, hek_X, handle_hs>)
    , Session_2($X, $TPM, h_hs, h_audit)
    , !Ek_sk_X($X, ek_sk_X)
]
--[
    TPM_getSessionAuditDigest()
    , Commit($TPM, $X, <'X_send_handles', hek_X, handle_hs>)
    , Running($TPM, $X, <'TPM_send_signature', auditInfo_signature>)
    , UniqueExecSign('TPM_signing')
    , TPM_Send_signed($TPM, auditInfo_hsession)
    , Secret(ek_sk_X)
]->
[
    Out(<$TPM, auditInfo_signature, auditInfo_hsession>)
]
```

5.2.3 Inter-Vfs Oblivious Remote Attestation protocol model

The agents interacting in the ORA protocol are:

- X: VF;

- TPM: X's trusted component (software);
- Y: VF that wants to verify the integrity of X.

The model of the ORA protocol is comprehensive of fifteen rules and, as the previous ones, starts with two initialization rules: the Platform_setup rule, that sets up the platform (TPM and X), similar to the one described for the Attestation Key creation protocol, and Y_init_and_send rule, that creates Y and generates a nonce, the challenge sent to X for attestation, and stores the public part of X's AK in the St_Y_Ak_pk_X linear fact, while the private part should not be compromised (its revealing is specified with the Ak_reveal rule).

In the Platform_setup rule the initial state consists in the declaration of what the platform knows: handles of keys, the PCRs, the nvPCRs. The functions used are:

- hak_get/0: function used to get the AK handle
- hek_pk_O_get/0: function used to get the Orchestrator's EK handle
- hi_get/0: function used to get the nvPCRs handle
- nv_read/1: function used to read the specific nvPCRs
- ak_get/1: function used to get the private part of the AK

The final state of this rule consists in having this data stored in persistent facts to be consumed when it is needed.

The restrictions used for modeling this protocol are the same ones used for the Attestation Key creation protocol.

```
rule Platform_setup:
let
    hak = hak_get()
    hek_pk_O = hek_pk_O_get()
    hi = hi_get()//handle per NVpcrs
    h_i = nv_read(hi) //NVPCR
    pcrs = pcrs_get()
    args = h(<h_i, '0', '0'>)
    ak = ak_get(hak)
in
[
]
--[
    Platform_setup($TPM, $X)
    , PlatformInit()
    , OnlyOnce()
    , OnlyOnceP('Platform')
    , Create($TPM)
    , Create($X)
    , Unique_Pairing($X), Unique_Pairing($TPM)
```

```
 ]->
[
    !St_pcrs($X, pcrs)
    , St_PlatformInit($TPM, $X)
    , !St_hak($X, hak)
    , !St_hek_O($X, hek_pk_O)
    , !St_hi_nv($X, hi, h_i)
    , !Ak($TPM, ak)
]

rule Y_init_and_send:
let
    nonce = ~n
    ak_pk_x = ak_pk_x_get()
in
[
    Fr(~n)
    , St_PlatformInit($TPM, $X)
]
--[
    Y_init()
    , Create($Y)
    , Running($Y, $X, <'Y_init', nonce>)
    ]->
[
    St_YInit($Y)
    , St_Y_Ak_pk_X($X, $Y, ak_pk_x)
    , St_Y_nonce($Y, nonce)
    , Out(<$Y, nonce>)
]
```

The protocol model starts with the X_send_sessionType rule, in which X takes the nonce sent by Y from the public channel with the In fact, and then generates and sends the session type to its TPM, that retrieves the session handle and sends it to X. The session handle retrieving is modeled in the TPM_StartAuthSession rule, using a function that accepts one parameter (session_type): sessionHandle_get/1.

In order to model the extension of the existing attestation policies to require proof that the specific agent handled the attestation, two rules are used: TPM_policyNV and TPM_PolicyPCRS. With this rules, after taking from the channel the handle of the session and the values of pcrs and nvpcrs sent by X, the TPM extends the policy and finally stores the hashed value in the Session_3 linear fact.

```
rule TPM_policyNV:
let
    args = h(<h_i, '0', '0'>)
    h_pol_new = h(<h_pol, 'CC_policyNV', args, name_hi(hi)>)
in
```

```
[
    In(<$X, hi, h_i, hps>)
    , Session_1($X, $TPM, hps, h_pol)
]
--[
    TPM_policyNV()
    , Eq(h_i,  nv_read(hi))
    , Commit($TPM, $X, <'X_send_forPolicyNV', <hi, h_i>>)]->
[
  Session_2($X, $TPM, hps, h_pol_new)
]

rule TPM_PolicyPCR:
let
    h_pcrs = <'0'>
    h_pcrs = <h_pcrs, pcrs>
    h_pol_policy_pcr = h(<h_pol_new, 'CC_policyPCR', pcrs, h(h_pcrs)>)
in
[
    In(<$X, pcrs, hps>)
    , Session_2($X, $TPM, hps, h_pol_new)
]
--[
    TPM_PolicyPCR()
    , Commit($TPM, $X, <'X_send_PCRS', pcrs>)]->
[
    Session_3($X, $TPM, hps, h_pol_policy_pcr)
]
```

The hash of the policy has to be compared to the one that X already knows (p). In order for this to happen, X starts by taking the values of p and t: in the X_send_for_policyAuthorize rule, correct values to p and t are given. Then, X sends them to the TPM using an Out fact, together with the Orchestrator name (hash of its public information).

```
rule X_send_for_policyAuthorize:
let
    p = h_pol_policy_pcr
    t = hmac(proof_t(),(<'VERIFIED', h(h_pol_policy_pcr)
        , h(ek_pub_O_get(hek_pk_O))>))
in
[
    !St_Hps_X($X, hps)
    , !St_hek_O($X, hek_pk_O)
    , Session_3($X, $TPM, hps, h_pol_policy_pcr)
]
--[
```

```
    X_send_for_policyAuthorize()
    , Running($X, $TPM, <'X_send_P_T', <p, t>>)]->
[
    Out(<$X, p, t, h(ek_pub_O_get(hek_pk_O)), hps>)
    , Session_4($X, $TPM, hps, h_pol_policy_pcr)
]
```

For modeling the TPM policy authorize phase, one rule is used: TPM_PolicyAuthorize. This rule describes the TPM checking the computed ticket and hashed policy against the values assumed authorized by the Orchestrator, p and t, which should have been authorized after the running of the Measurement update protocol), by means of the Equality restriction. After the verification, X has to send the challenge(nonce), previously received by Y, consuming the linear fact St_nonce(nonce) and producing an Out fact with that specific value for the TPM signing process. The TPM_sign rule models the signing process of the challenge: the TPM signs the nonce using the sign function (function that it is defined in the signing built-in) with its AK and puts the signature in the public channel for X to forward it to Y (X_send_to_Y rule) for verification.

```
rule TPM_sign:
let
    signature = sign(nonce, ak)
in
[
    In(<$X, nonce, hak, hps>)
    , Session_5($X, $TPM, hps, h_pol_policyAuthorize)
    , !St_auth_pol($TPM, auth_pol)
    , !Ak($TPM, ak)
]
--[
    TPM_sign(), Eq(h_pol_policyAuthorize, auth_pol)
    ,UniqueExecSign('TPM_signing')
    , Commit($TPM, $X, <'X_send_for_signing', nonce>)
    , Running($TPM, $X, <'TPM_send_signature', signature>)
    , Send_signed($TPM, signature)
  ]->
[
    Out(<$TPM, signature>)
]
```

The last rule that models the ORA protocol is the Y_verify_signature rule: this rule describes Y taking from the public channel the signature over the nonce done by the TPM of the other virtual function and also taking the nonce generated by itself consuming the St_Y_nonce linear fact. The verification of the signature and the nonce are modeled using the Equality restriction and the verify function, provided by the signing built-in.

```
rule Y_verify_signature:
[
    In(<$X, signature>)
    , St_Y_nonce($Y, nonce)
    , St_Y_Ak_pk_X($X, $Y, ak_pk_x)
]
--[
    Y_verify_signature()
    , Honest($X)
    , Honest($TPM)
    , Honest($Y)
    , Authentic($TPM, signature)
    , Eq(verify(signature, nonce, ak_pk_x),true)
    , Commit($Y, $X, <'X_send_signature_to_Y', signature>)]->
[]
```

5.2.4 Attaching NVPCRs protocol model

The entities involved in the Attaching NVPCRs protocol are three:

- O: Orchestrator;
- X: Virtual Function;
- TPM: trusted component (software).

For the model of this protocol, after the usual Platform_setup rule and Orc_init rule, the rule NV_true has been written. This rule is the starting rule of the model of the Attaching protocol, because it starts the process creating the req_add message, which contains all the information needed for the creation and extension of the new NVPCR:

- The Identifier;
- The template: it describes the attributes of the register;
- Authorization policy: policy that can be used by the Orchestrator to delete the register;
- IV: an initial value to extend.

```
rule NV_true:
let
    iv = '0'
    idx = idx_get()
    name_O = name_Orc()
    TPL_idx = TPL_get(idx)
    nv = 'true'
```

```
    hpol = h(<h(h(<'00000000', 'CCpolicyauthorize', name_O>))
        , 'CCpolicycommandcode','CCnvundefinespacespecial'>)
    req_add = <idx, TPL_idx, hpol, iv, nv>
in
[
]
--[
    NV_true()
    , Running($O, $X, <'send_req_add', req_add>)
]->
[
    Out(<$O, req_add>)
]
```

X takes these values from the public channel and forwards them to its TPM, which then execute a series of commands:

- NV_DefineSpace;
- NV_Extend;
- NV_Certify.

What the TPM does with these three commands has been modeled in three specific rules: TPM_NV_defineSpace, TPM_NV_Extend, TPM_NV_Certify.

The first of these three rules is needed to model the creation of the space for the NVPCR: this is done using a private function called nvCreate/4, which accepts as parameters the identifier, the handle of the secret platform primary seed, the template with the attributes and the policy created by the Orchestrator. After the TPM_NV_defineSpace rule, the TPM_NV_Extend rule extends the initial value (IV) sent by the Orchestrator, writing the newly created NVPCR. The last action modeled, done by the TPM, is specified in the TPM_NV_Certify rule: the TPM certifies the new NVPCR, signing the certificate information with the secret part of X's Endorsement Key, stored in the !Ek_sk_X persistent fact.

```
rule TPM_NV_defineSpace:
let
    nv_create = nvCreate(idx, hpps, TPL_idx, hpol)
in
[
    In(<$X, hpps, idx, TPL_idx, hpol>)
]
--[
    TPM_NV_defineSpace()
    , Commit($TPM, $X, <'send_hpol', hpol>)
  ]->
[
    St_TPM_template_hpol($TPM, TPL_idx, hpol)
```

```
]

rule TPM_NV_Extend:
let
    nv_write = h(<nvRead(idx), iv>)
in
[
    In(<$X, idx, iv>)
]
--[
    TPM_NV_Extend()
    , Commit($TPM, $X, <'send_iv', iv>)
]->
[
    St_TPM_iv_idx($TPM, iv, idx)
]

rule TPM_NV_Certify:
let
    certInfo_magic = 'TPM_GENERATED'
    certInfo_nvContents = h(<'00000000', iv>)
    certInfo_objName = <h(TPL_idx), <'WRITTEN', idx, hpol>>
    certInfo = <certInfo_magic, certInfo_nvContents, certInfo_objName>
    certInfo_signature = sign(certInfo, ek_sk_X)
in
[
    In(<$X, hpps, idx, hek_X>)
    , !Ek_sk_X($TPM, ek_sk_X)
    , St_TPM_iv_idx($TPM, iv, idx)
    , St_TPM_template_hpol($TPM, TPL_idx, hpol)
]
--[
    TPM_NV_Certify()
    , Commit($TPM, $X, <'send_handles', hpps>)
    , Running($TPM, $X, <'TPM_send_certInfo', certInfo>)
    , UniqueExecSign('TPM_signing')
    , TPM_Send_signed($TPM, certInfo)
    , Secret(ek_sk_X)
  ]->
[
    Out(<$TPM, $TPM, certInfo_signature, certInfo>)
]
```

The end of the protocol consists in the Orchestrator verifying the signature and the correctness of the certificate information, checking the values against correct ones. For the verification to be correct, the certInfo fields have been given the correct values in the TPM_NV_Certify rule.

```
rule Orc_verify:
let
    certInfo = <certInfo_magic, certInfo_nvContents, certInfo_objName>
in
[
    In(<$X, certInfo_signature, certInfo>)
    , St_TPM_iv_idx($TPM, iv, idx)
    , St_TPM_template_hpol($TPM, TPL_idx, hpol)
]
--[
    Orc_verify()
    , Commit($O, $X, <'X_send_to_Orc', certInfo>)
    , Eq(verify(certInfo_signature, certInfo,
            ek_pk_get(handle_ek_X_get)), true)
    , Eq(certInfo_magic, 'TPM_GENERATED')
    , Eq(certInfo_nvContents, h(<'00000000', iv>))
    , Eq(certInfo_objName, <h(TPL_idx), <'WRITTEN', idx, hpol>>)
    , Authentic($TPM, certInfo)
    , Honest($X)
    , Honest($TPM)
    , Honest($O)
]->
[
]
```

5.2.5 Detatching NVPCRs protocol model

The entities involved in the Detatching NVPCRs protocol are three:

- O: Orchestrator;
- X: Virtual Function;
- TPM: trusted component (software).

The Detatching NVPCR protocol model starts with the initialization of the Platform and the Orchestrator, storing in persistent facts all the key handles useful. The rule that starts the protocol is TPM_StartAuthSession_1, in which the TPM generates a fresh nonce with the Fr fact, and sends it to X, that forwards it to the Orchestrator. The nonce generated is used by the Orchestrator in the NV_true rule: the Orchestrator creates the ahash value, which is a digest needed to authorize a policy hpol, done over the nonce and the hcp value. With these values the Orchestrator creates the request for deletion (req_delete) and with an Out fact sends it to X. Moreover, the St_Orc_reqDel stores the request.

```
rule NV_true:
let
```

```
    hpol = h(h(<'00000000', 'CCpolicysigned', name_Orc()>))
    hhpol = h(hpol)
    signhhpol = sign(hhpol, ek_sk_O)
    hcp = h(<'CCnvundefinespacespecial', idx>)
    ahash = h(<n, '00000000', hcp>)
    signahash = sign(ahash, ek_sk_O)
    req_delete = <idx, hcp, signahash, hpol, hhpol, signhhpol>
in
[
    In(<$X, n>)
    , !St_idx($O, idx)
    , !St_Orc_handle($O, hek_O)
    , !St_Ek($O, ek_sk_O)
]
--[
    NV_true()
    , Commit($O, $X, <'X_forward_n', n>)
    , Running($O, $X, <'send_req_del', req_delete>)
    , Orc_send_reqDel_withSignature($O, signhhpol)
    ]->
[
    Out(<$O, req_delete>)
    , St_Orc_reqDel($O, req_delete)
]
```

X sends the session type to the TPM (X_receives_req_del_nv_true rule), which starts the session in the TPM_StartAuthSession rule. After starting the session, X sends req_del values to the TPM. The TPM executes a series of commands, modeled with rules that have the same name of the commands. These commands are:

- TPM_VerifySignature;
- TPM_PolicySigned;
- TPM_PolicyAuthorize;
- TPM_PolicyCommandCode;
- TPM_NV_UndefineSpaceSpecial.

In the TPM_VerifySignature rule is modeled the TPM verifying the Orchestrator's signature over the policy. The signature verification is done by using the Equality restriction. The TPM in this rule also construct the ticket, in order to attest that the signature has been verified.

```
rule TPM_VerifySignature:
let
    ticket = hmac(proof_get(), <'VERIFIED', hhpol, name_Orc()>)
in
```

```
[
    In(<$X, hhpol, signhhpol, handle_ek_Orc>)
]
--[
    TPM_VerifySignature()
    , Commit($TPM, $X, <'X_send_h_hpol', hhpol >)
    , Running($TPM, $X, <'TPM_sends_ticket', ticket >)
    , Eq(verify(signhhpol,hhpol,ek_sk_O_get(handle_ek_Orc)),true)
    , Honest($TPM)
    , Honest($X)
    , Honest($O)
    , Authentic($O, signhhpol)
]->
[
    Out(<$TPM, ticket>)
]
```

The TPM_PolicySigned rule models the TPM verifying the other signature done by the Orchestrator, the one over the aHash value. For the verification to be correct, the aHash value has been given the correct value. The final state of this rule, stores in the Session linear fact the new policy.

```
rule TPM_PolicySigned:
let
    ahash = h(<nonce, '00000000', hcp>)
    h_pol_new = h(h(<'00000000', 'CCpolicysigned', name_Orc()>))
    cpHash = hcp
in
[
    In(<$X, signahash, hcp, nonce, hek_O_pk, hps>)
]
--[
    TPM_PolicySigned()
    , Commit($TPM, $X, <'X_send_for_policySigned', hps >)
    , Eq(verify(signahash,ahash,ek_pk_O_get(hek_O_pk)),true)
]->
[
    Session(h_pol_new, cpHash)
]
```

TPM_PolicyAuthorize rule verifies the ticket previously constructed and modifies the value of the policy to authorize it.

```
rule TPM_PolicyAuthorize:
let
```

```
    ticket = hmac(proof_get(), <'VERIFIED', h(h_pol_X), name_O>)
    h_pol = h(<'00000000', 'CCpolicyauthorize', name_O>)
in
[
    In(<$X, h_pol_X, t_X, name_O, hps>)
    , Session(h_pol_new, cpHash)
]
--[
    TPM_PolicyAuthorize()
    , Commit($TPM, $X, <'X_send_for_policyAuthorize', h_pol_X >)
    , Eq(h_pol_X, h_pol_new)
    , Eq(ticket, t_X)
]->
[
    Session_2(h_pol, cpHash)
]
```

The TPM_PolicyCommandCode rule models the TPM restricting the session's command code and storing the new policy value.

```
rule TPM_PolicyCommandCode:
let
    cc = cc_nv_undefineSpaceSpecial
    h_pol_undefineSpecial = h(<h_pol, 'CCpolicycommandcode', cc>)
in
[
    In(<$X, hps, cc_nv_undefineSpaceSpecial>)
    , Session_2(h_pol, cpHash)
]
--[
    TPM_PolicyCommandCode()
    , Commit($TPM, $X,
            <'X_send_for_policyCommandCode',
            cc_nv_undefineSpaceSpecial>)
]->
[
    Session_3(h_pol_undefineSpecial, cpHash, cc)
]
```

The last rule, TPM_nv_undefineSpaceSpecial, models the TPM deleting the NVPCR, if the authorization policy is fulfilled.

```
rule TPM_nv_undefineSpaceSpecial:
let
    authPol = h(<h(<'00000000', 'CCpolicyauthorize', name_Orc>)
```

```
                , 'CCpolicycommandcode', cc>)
    destroy = nv_and_hps_destroy()
in
[
    In(<$X, hpps, hps, idx>)
    , Session_3(h_pol_undefineSpecial, cpHash, cc)
]
--[
    TPM_nv_undefineSpaceSpecial()
    , Commit($TPM, $X, <'X_send_for_nv_undefineSpaceSpecial', hps >)
    , Eq(authPol, h_pol_undefineSpecial)
    , Eq(cc, 'CCnvundefinespacespecial')
    , Eq(cpHash, h(<'CCnvundefinespacespecial', idx>))
]->
[
]
```

Chapter 6

Results

In this chapter the results given by Tamarin prover on the symbolic models described before are presented. Tamarin prover is based on the use of lemmas. Lemmas represents the security properties that have to be verified for the model. The framework analyses the model until it can prove that a property holds or finds a counterexample.

6.1 Attestation by quote verification

The attestation by quote protocol has been modeled as described in Section 5.1. The interaction between the TPM and the Fog Node is analysed using a public channel (Section 2.4.4).

6.1.1 Attestation by quote: Lemmas

The security properties examined for all the protocols are modeled with the lemmas of:

- Functional correctness;
- Message authentication;
- Aliveness;
- Injective agreement.

In the next section the lemmas will be explained, which are the same for all the protocols, reporting in particular the code of those of the Attestation by Quote.

Functional Correctness

This lemma is used to check the correct flow of execution of the protocol. It checks if the actions are performed in the correct order; it also takes into account two action facts:

- UniqueExecUpdateM: it is an action fact of the TPM_update_measure rule. In the lemma is used in order to check that the update operation is executed only once at the same moment in time;

- UniqueExecSign: it is an action fact of the TPM_signAndSend rule. In the lemma is used in order to check that the signing operation is executed only once at the same moment in time.

```
lemma update_and_attestation_functional_correctness:
exists-trace
"Ex #a #b #c #d #e #f #g #i #l #m  #o #n.
/*does not exist an endorsement key reveal*/
not( Ex C #k. Reveal_ek(C) @k) &
PlatformInit() @ a &
Orc_init() @ b &
Tracer_init() @ c &
Orc_send_update_req() @ d &
Tracer_update()@ e &
F_update_measure() @ f &
TPM_update_measure() @ g &
O_send_nonce() @ i &
F_send_nonce_to_TPM() @ l &
TPM_signAndSend() @ m &
FogNode_sendSigned() @ n &
Orc_receiveSigned_verify() @ o &
a < b &
b < c &
c < d &
d < e &
e < f &
f < g &
g < i &
i < l &
l < m &
m < n &
n < o &
( All #i #j x. UniqueExecUpdateM(x) @i & UniqueExecUpdateM(x) @j
==> #i = #j) &
( All #i #j x. UniqueExecSign(x) @i & UniqueExecSign(x) @j
==> #i = #j)
"
```

Message authentication

This lemma is used to verify that the message containing the quote certificate is sent by the TPM through an honest Fog Node and received by the Orchestrator without any not Honest agent revealing the TPM's secret key.

```
lemma message_authentication:
"All f m #i. Authentic(f,m) @i
```

```
==> (Ex #j. Send_sig(f,m) @j & j<i)
| (Ex B #r. Reveal_ek(B)@r & Honest(B) @i & r < i)"
```

Injective agreement

This lemma models the unique chain of events driven by specific input (nonce, I) and guarantees that an agent A has a unique matching partner B in the interaction.

```
lemma injectiveagreement:
"All A B t #i.
Commit(A,B,t) @i
==> (Ex #j. Running(B,A,t) @j
& j < i
& not (Ex A2 B2 #i2. Commit(A2,B2,t) @i2
& not (#i2 = #i)))
| (Ex C #r. Reveal_ek(C)@r & Honest(C) @i)"
```

Aliveness

This lemma is used to guarantee to an agent A aliveness of another agent B: whenever A completes a run of the protocol, then B has previously been running the protocol.

```
lemma aliveness:
"All a b t #i.
Commit(a,b,t)@i
==> (Ex #j. Create(b) @ j)
| (Ex C #r. Reveal_ek(C) @ r & Honest(C) @ i)"
```

6.1.2 Attestation by quote: Results

The functional correctness lemma for this protocol is verified: this means that the model holds. Proving the property described by the message authentication lemma, however, Tamarin finds vulnerabilities. From the execution of the model appears that the Attacker, during the update phase, can modify the id of the configuration file (Figure 6.1) and also the hash of the file (Figure 6.2) and send it in the channel pretending to be the Tracer.

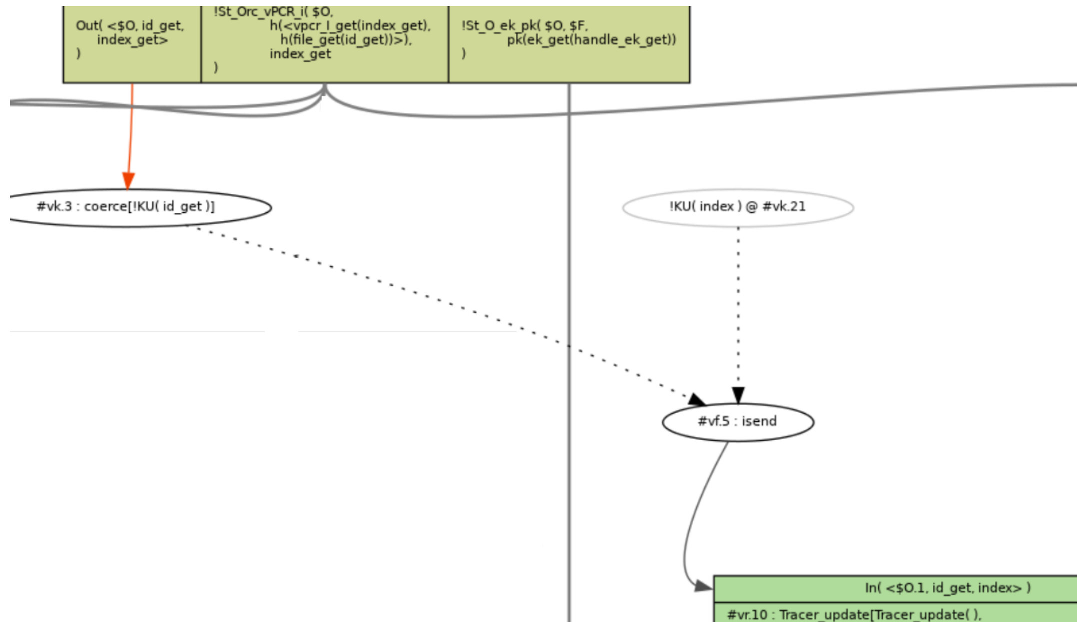

Figure 6.1: Attestation by Quote protocol - Attacker coercing id of the configuration file and index.

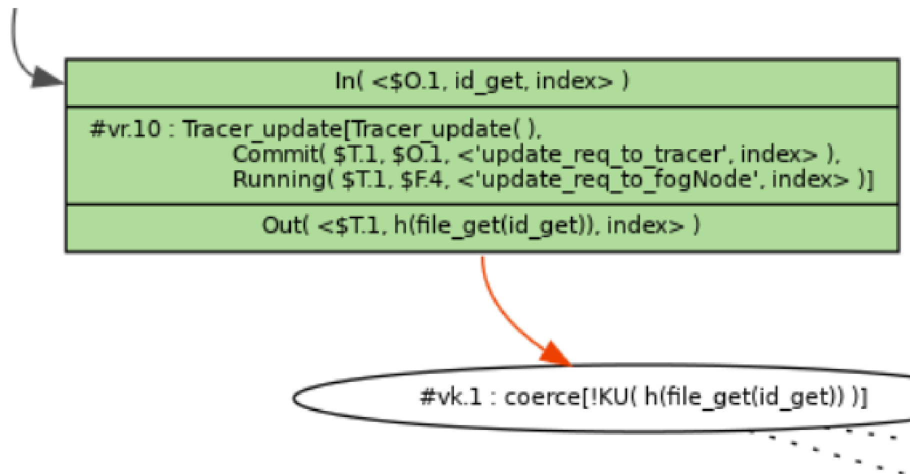

Figure 6.2: Attestation by Quote protocol - Attacker coercing the hash of the configuration file.

During the attestation phase it can send a different nonce pretending to be the Orchestrator (Figure 6.3), manipulate the message containing the signature of the quote (Figure 6.4) and the quote. The Attacker can inject messages directly on the channel: it can pretend to be each of the entities, so sending the ids, hash of the file and nonce, it can construct the quote with the values inserted before and make the process of attestation happen without the correct data. Even if the Orchestrator discards the message, which is not verified, the Attacker can perform a DoS attack.

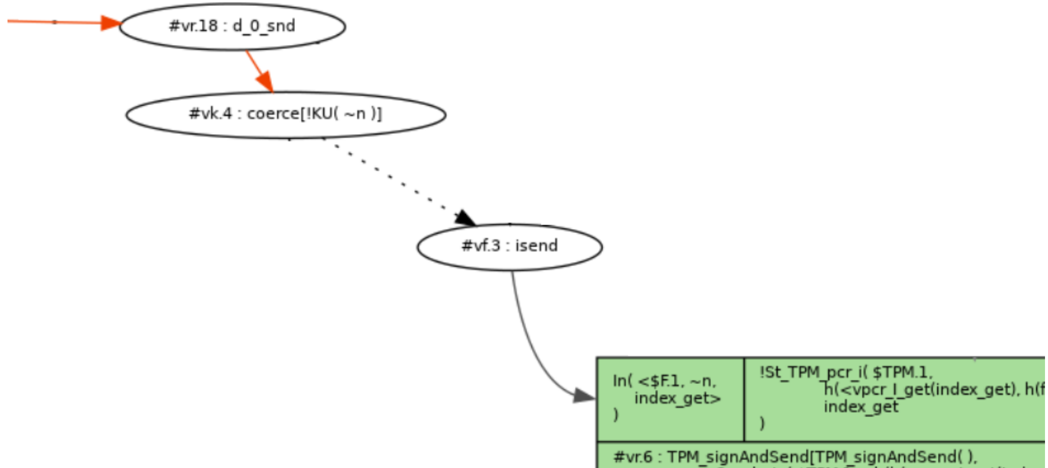

Figure 6.3: Attestation by Quote protocol - Attacker coercing nonce.

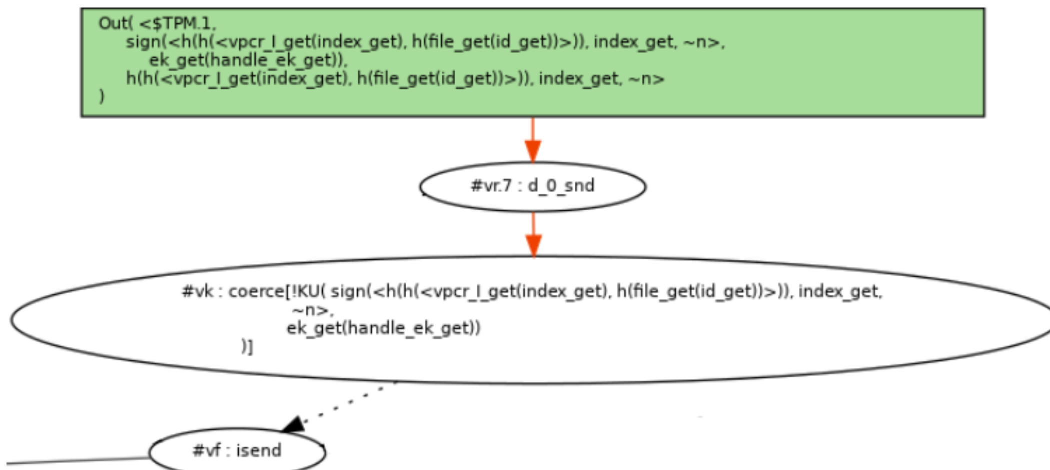

Figure 6.4: Attestation by Quote protocol - Attacker coercing the signing message.

A possible attack that the Attacker can perform knowing the signature message, is also the DoS attack: even if the Orchestrator checks the signature and discards it if it is not verified, the attack can have impact on the performance. This problem is given by the fact that this scenario does not consider an authentication protocol (peer and message authentication).

The last two lemmas (Aliveness and Injective agreement) are also not proved correct, as a consequence of the absence of a peer authentication protocol. The Attacker can pretend to be both the Orchestrator and the Tracer, making the interaction with the platform (TPM and Fog Node) start and continue without the real entities. The Figure 6.5 shows the traces found for these lemmas by Tamarin.

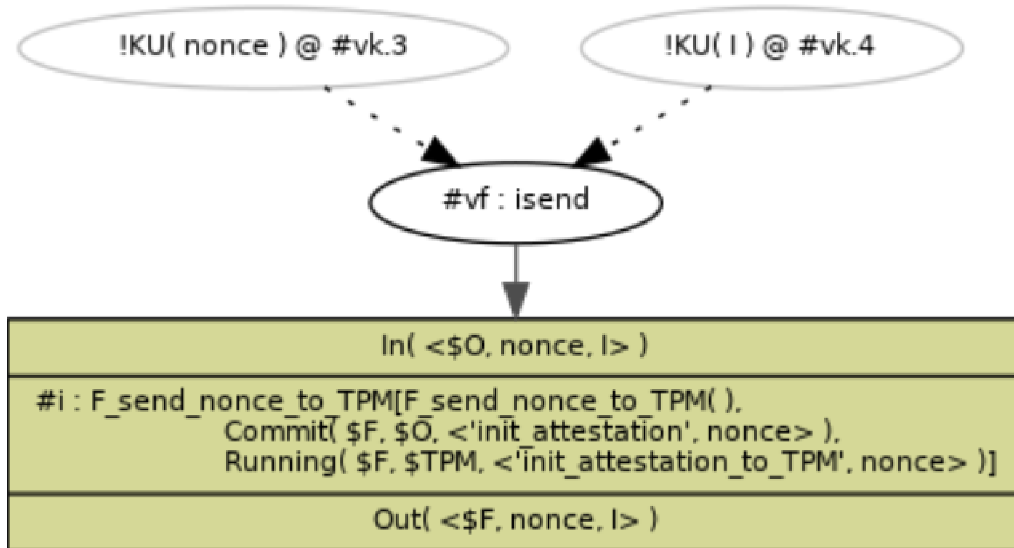

Figure 6.5: Attestation by Quote - Aliveness and Injective agreement result.

6.2 Oblivious Remote Attestation verification

The Attestation Key creation protocol, the Measurement update protocol and the ORA one have been modeled as described in Section 5.2, as well as the two protocols describing the Attaching and Detaching mechanisms of the NVPCRs (Section 3.2.4). Interactions between agents takes place on a public channel. The lemmas proved for these protocols are the same as the ones of the Attestation by Quote protocol, but written with action facts that are specific for them.

6.2.1 Attestation Key creation: Results

The verification of Attestation Key Creation protocol highlights some possible vulnerabilities, but apart from a remote case and the possibility of DoS attack, the traces do not represent a threat to the system.

The functional correctness lemma is proved verified: the model is consistent and exist a trace of the protocol following the correct flow. Despite this, it is interesting to note that the attacker can manipulate everything on the public channel, and can pretend to be the Orchestrator, making the execution of the protocol start; it can send X messages that should come from either the Orchestrator and TPM, or send messages pretending to be the TPM. However, the Orchestrator does verify the content of the messages and the signature at the end of the protocol, hence the attacker's access points do not compromise the flow of execution, but a possible problem can be a DoS attack: even if the Orchestrator discards problematic messages, if it receives a great number of them, performance can deteriorate.

Another vulnerability is shown in the Figure 6.6 below. During the execution, the attacker can manage to take control of the TPM, compromising all the actions that have to be done by the platform, in particular the creation of the AK. If the attacker has control of the TPM, it can have access to the endorsement key, construct the certificate and sign it: Tamarin shows how the message containing the signature, in this case, it is not the authentic one. The case of the Attacker taking control of the

TPM, however, is a very remote case, so even if the lemma is not valid in all the traces, it cannot be considered as a possible attack.

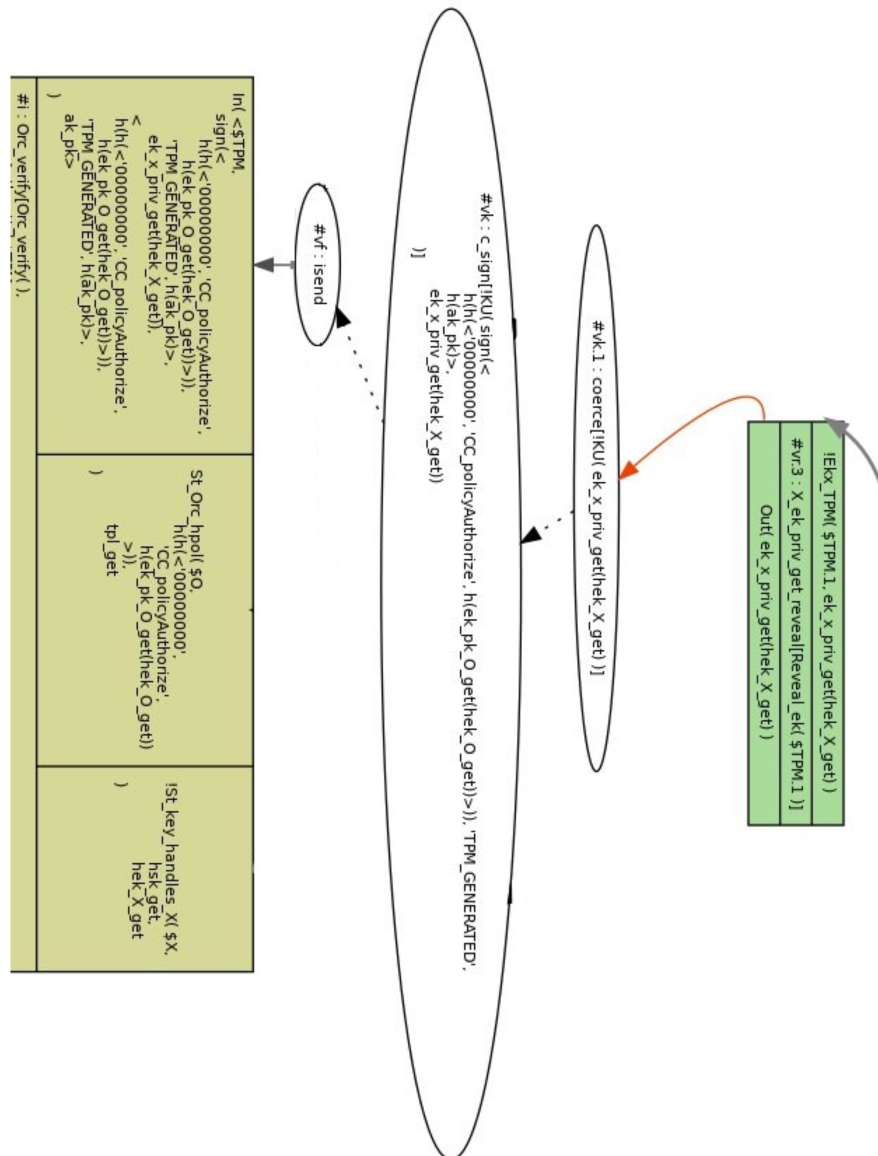

Figure 6.6: Attestation Key creation protocol - Message Authentication result.

The Figure 6.7, shows how the attacker, having access to the public channel, can probably construct some messages, such as attestation key, ticket and the hash of the creation details of the ak, and insert them in the channel, sending them to X as they are coming from the TPM, compromising X. This operation indicates that the properties of aliveness and injective agreement are not proved, because X can start a series of actions without the warranty of having another agent running the protocol and of having a unique matching partner in the interaction: X acts in response to the attacker and not to a real agent. Analysing this trace, it is possible to notice that the one possible attack could be a DoS attack, while any other attack would

not be possible, because at the end of the protocol the Orchestrator always verifies the certificate and discards any wrong message, marking the platform as untrusted.

Figure 6.7: Attestation Key creation protocol - Aliveness and Injective agreement.

6.2.2 Measurement update: Results

The Measurement update protocol is based on the interactions between TPM, X, Orchestrator and the Tracer on a public channel. Lemmas written for the model of this protocol are not all verified. Aside from the functional correctness lemma, which also in this case shows the correctness of the model's flow of execution, the results given by Tamarin for the other lemmas highlight some possible problems, but the traces found do not represent a compromise of the system.

One possible problem could be if the attacker manages to extract the second term of the Out fact of the Orc_update_request rule, which is the update request: it can then coerce it and its signature and send them directly to the TPM for the VerifySignature command (Figure 6.8). This trace proves that the message arriving to TPM can be compromised and not be authentic. In the same way the attacker can manipulate the signature of the audit information of the session and send to the Orchestrator an unauthentic message: this interference is shown in the Figure 6.9. These two interferences, however, do not compromise the protocol: the first one is managed by the VerifySignature command, where if the message is not verified it will be discarded; the second one is managed by the Orchestrator at the end of the protocol, which will discard any unverified messages.

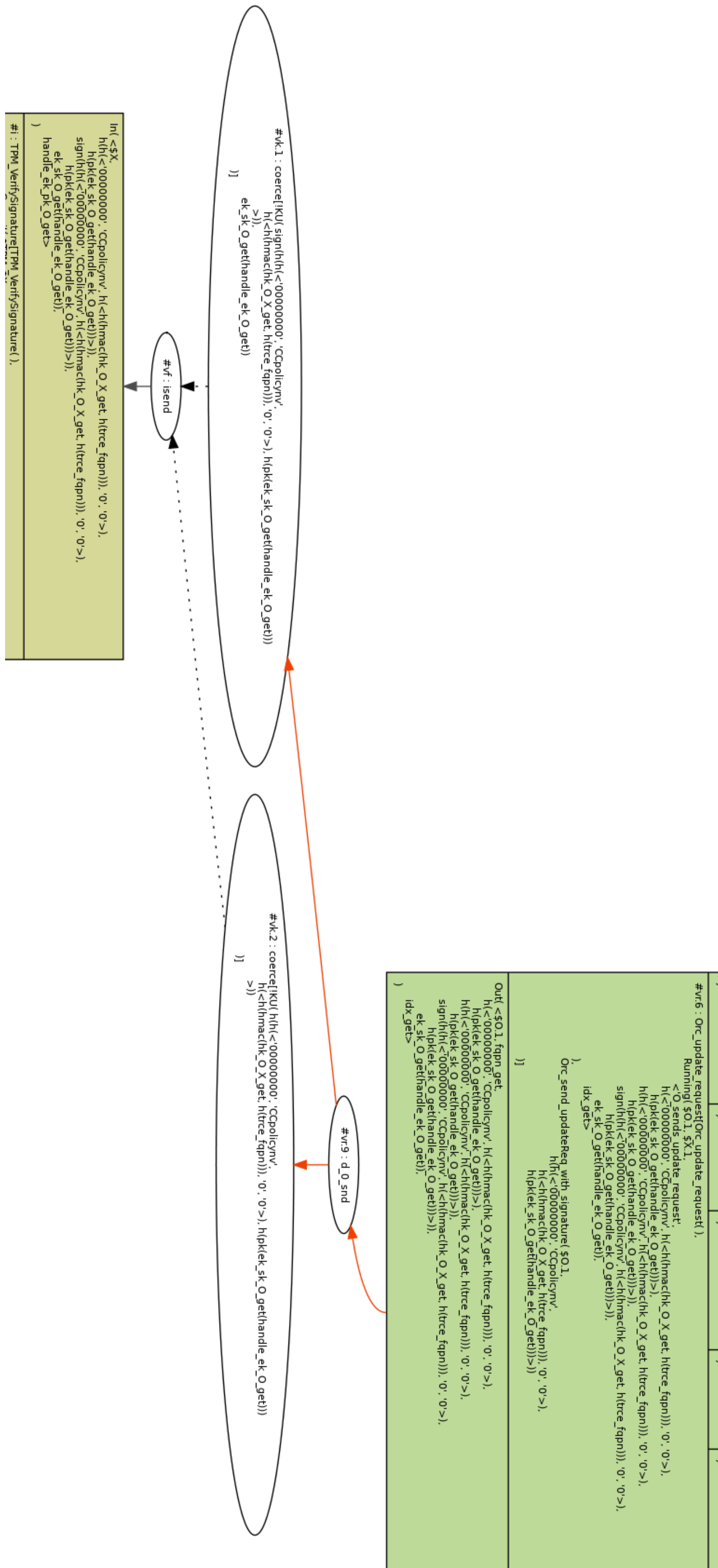

Figure 6.8: Measurement Update protocol - Message authentication - 1.

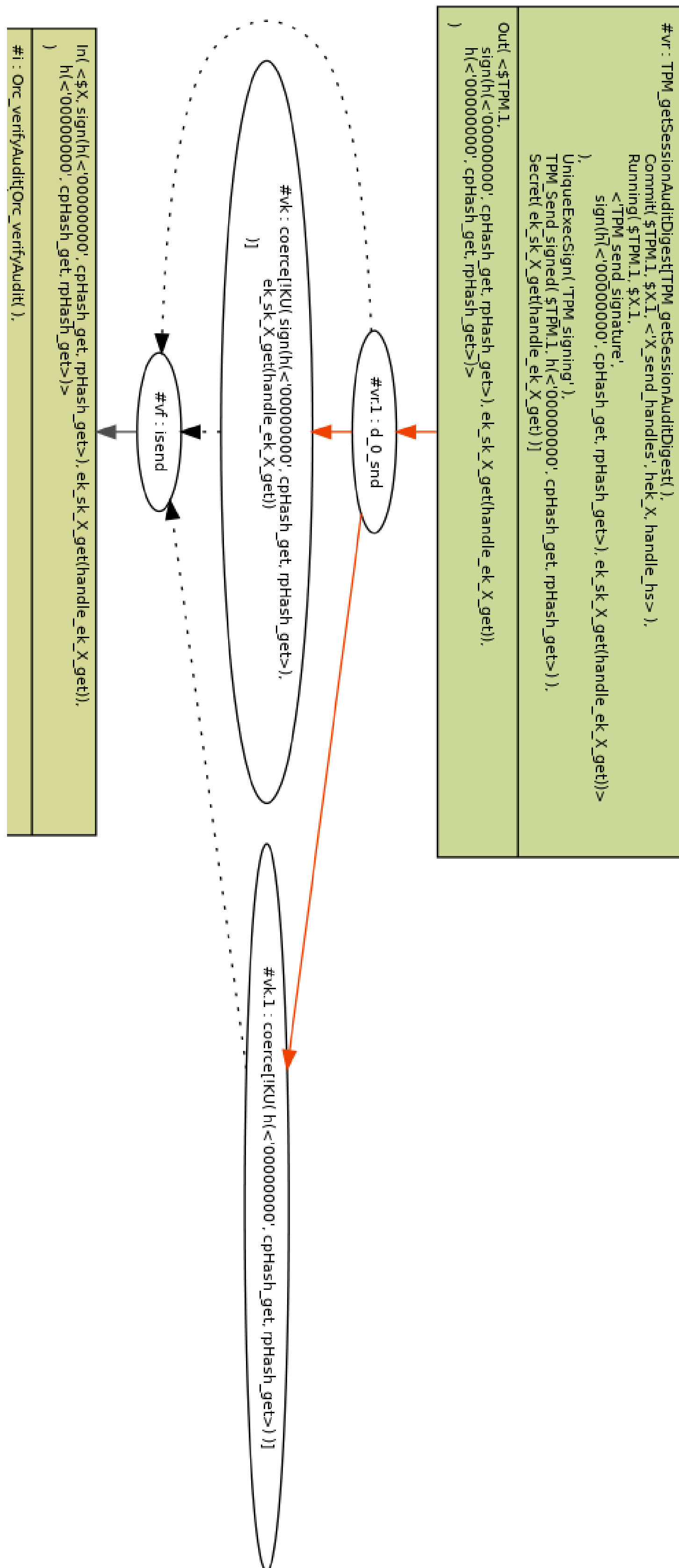

Figure 6.9: Measurement Update protocol - Message authentication - 2.

Regarding Aliveness and Injective agreement, results are shown in the Figure 6.10. The figure shows, in particular, how the attacker, pretending to be the Tracer, can send a message to the Orchestrator, even if the Orchestrator is not running the protocol. However, this does not compromise the system.

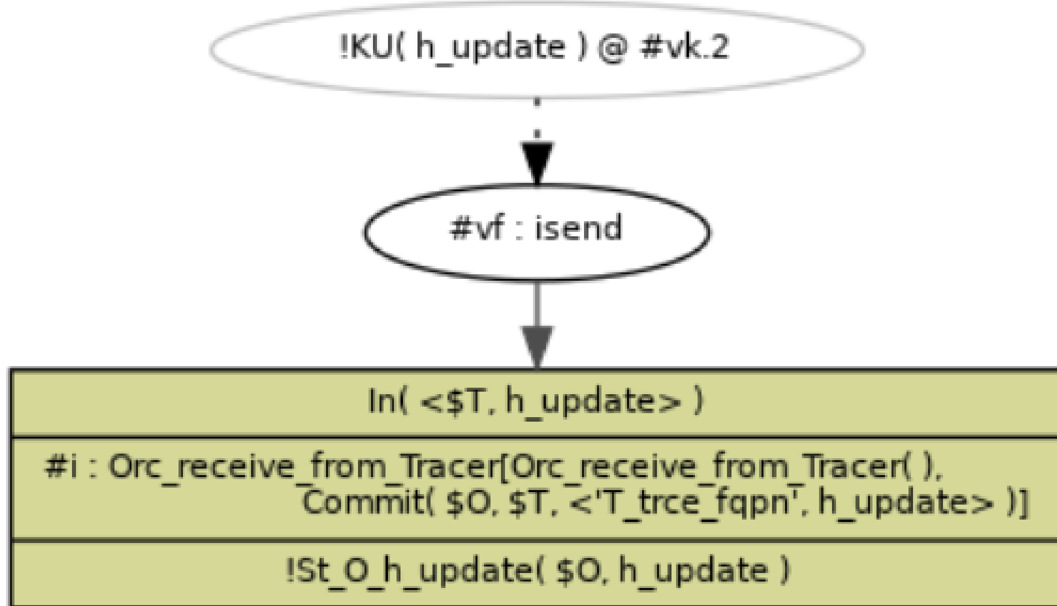

Figure 6.10: Measurement Update protocol - Aliveness.

6.2.3 Oblivious Remote Attestation: Results

The Oblivious Remote Attestation protocol sees the interaction of two virtual functions, X and Y, and the TPM, with Y trying to verify the integrity of X, as explained in the Section 3.2.3. The functional correctness lemma for the ORA protocol is verified, this means that exists a trace in which the model follows the correct flow of execution. As the results of the previous protocols, also this one shows how the attacker can interact with the involved agents(in this case X, Y and TPM) without an authentication mechanism: even if the message that are not correct are discarded after the verification, the attacker can pretend to be any of the agents and perform a DoS attack.

From the Figure below (Figure 6.11) it emerges that the attacker can start the execution of the protocol, sending the message containing the session type, directly to the TPM, pretending to be X. This action can make the TPM start the execution of the protocol, in particular executing the startAuthSession command, without other agents running the protocol and without having a trusted partner in the interaction, but interacting with the adversary that pretends to be X. The same trace is shown from the injective agreement lemma: this is because the startAuthSession command can be executed by the TPM without having a unique partner (X) in the interaction. However, no possible attack can arise from these traces.

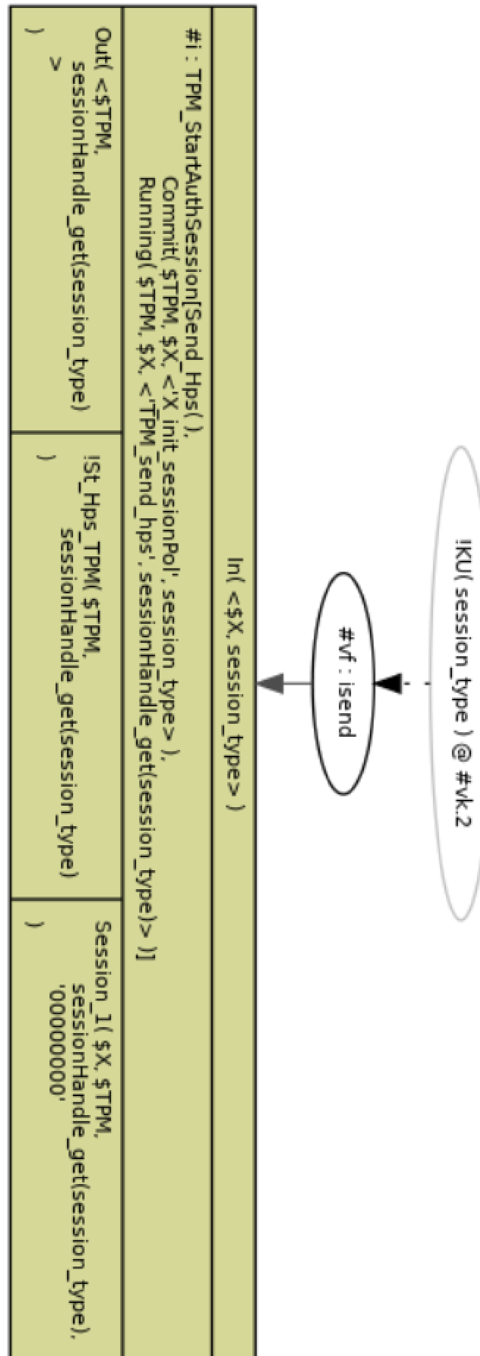

Figure 6.11: ORA protocol - Aliveness and Injective agreement.

A possible trace of attack found by Tamarin prover is shown in the Figure 6.12. The figure describes how the attacker, if it manage to take control of the TPM, can coerce the nonce, which is sent in clear on the channel by Y to X as a challenge for integrity attestation, sign it with the attestation key and send it to Y for verification, pretending to be X. However, this trace describes a remote case, the one of the attacker having full control over the TPM.

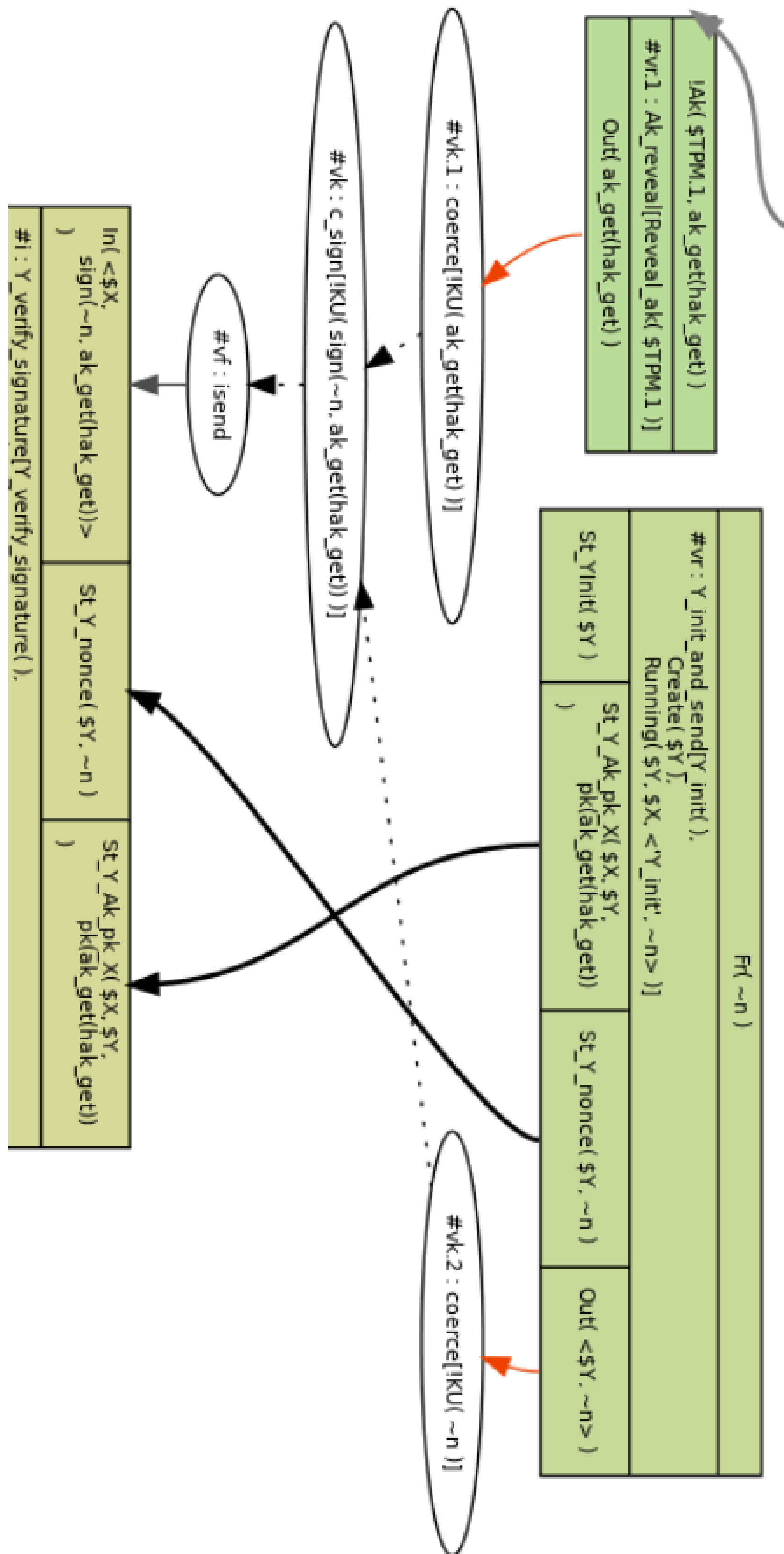

Figure 6.12: ORA protocol - Message authentication.

6.2.4 Attaching NVPCRs: Results

The analysis of the Attaching NVPCRs protocol model shows some problems. Particularly, from the trace found by Tamarin for the message authentication lemma, it emerges that the Attacker can manipulate a great number of messages. Starting from the Orchestrator inserting in the public channel the add request, if the attacker gets to know it(d_0_snd rule), can coerce the parameters of the request: the identifier, the template and the hash of the authorization policy, as it is shown in Figure 6.13. Knowing these information and constructing the others, the Attacker manage to send them to the TPM in order to start the NV_nvExtend command, the NV_defineSpace one, and the NV_certify one, sending information instead of X, multiple times. Doing so, the Attacker pretends to be X and, after making the TPM certify the new corrupted PCR, it can intercept, extract and coerce the signature (Figure 6.14): this means that the message received by the Orchestrator containing the signature of the certificate and the certificate can come from the attacker and not from X. These trace highlights a possible attack: the Attacker can make the TPM executes its commands multiple times, always changing the identifier of the NVPCR to be created. Acting like this, can possibly lead to the creation of multiple unauthorized NVPCRs and memory saturation.

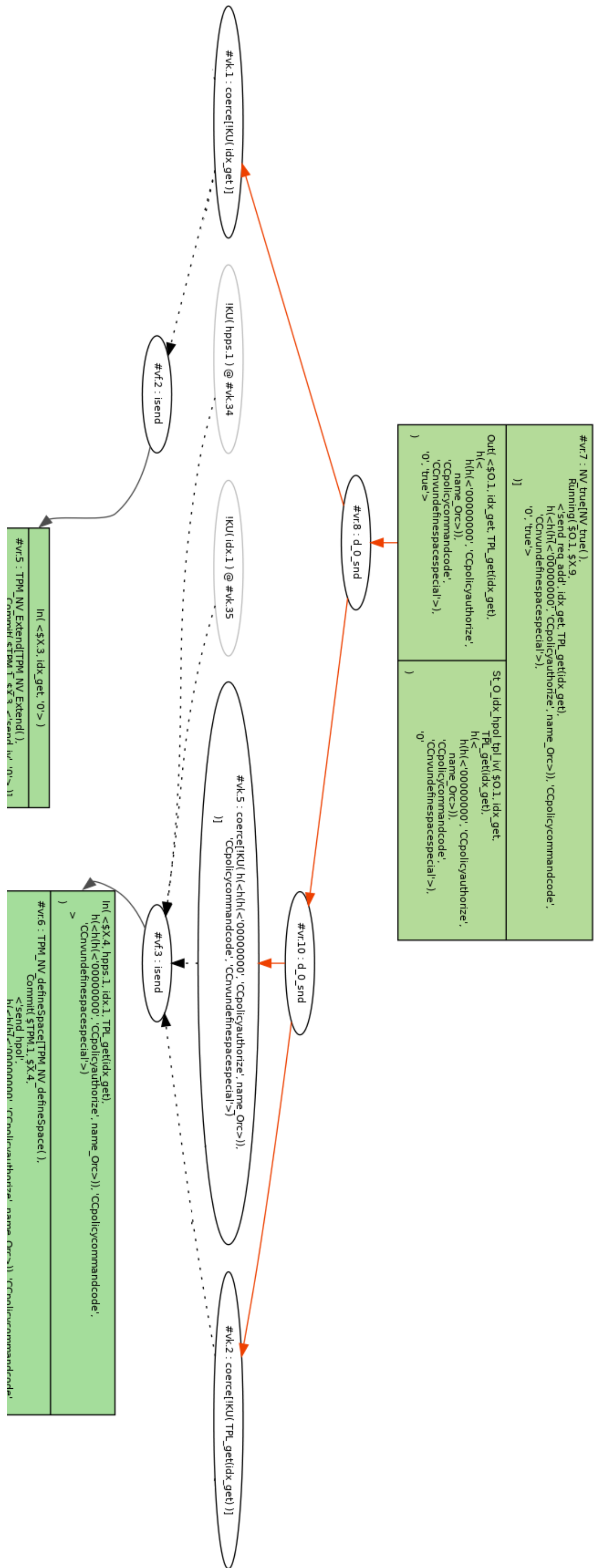

Figure 6.13: Attaching NVPCRs protocol - Message Authentication - 1.

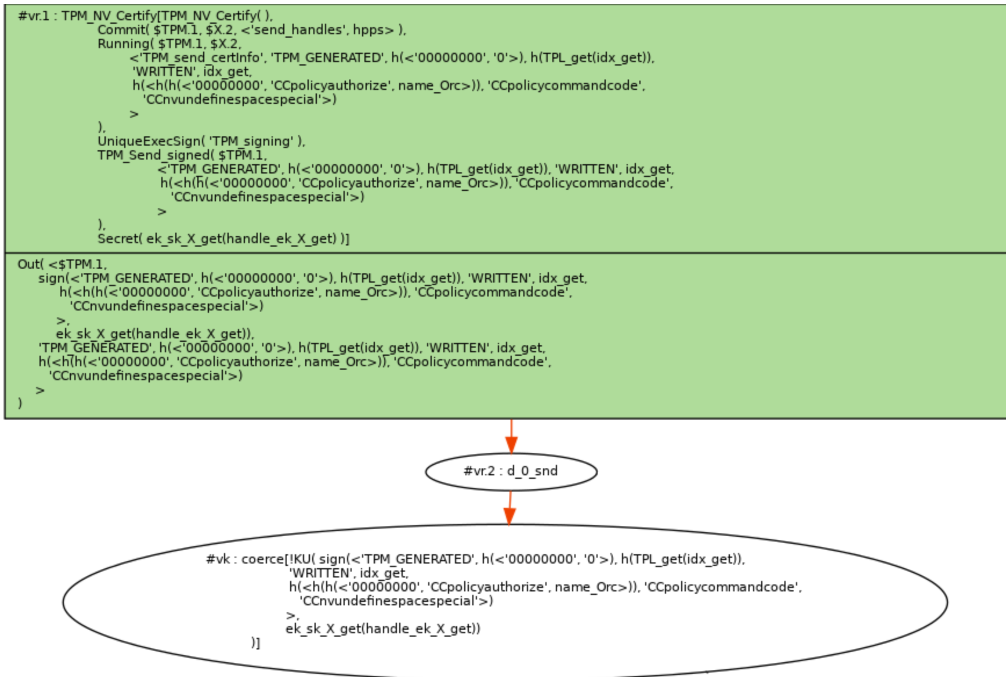

Figure 6.14: Attaching NVPCRs protocol - Message Authentication - 1.

The Figure 6.15 below, shows the trace given by Tamarin for the Aliveness and Injective Agreement lemmas for this protocol. The result shows how the Attacker can make the TPM execute the NV_Extend command without the Orchestrator starting the execution of the protocol with its add request.

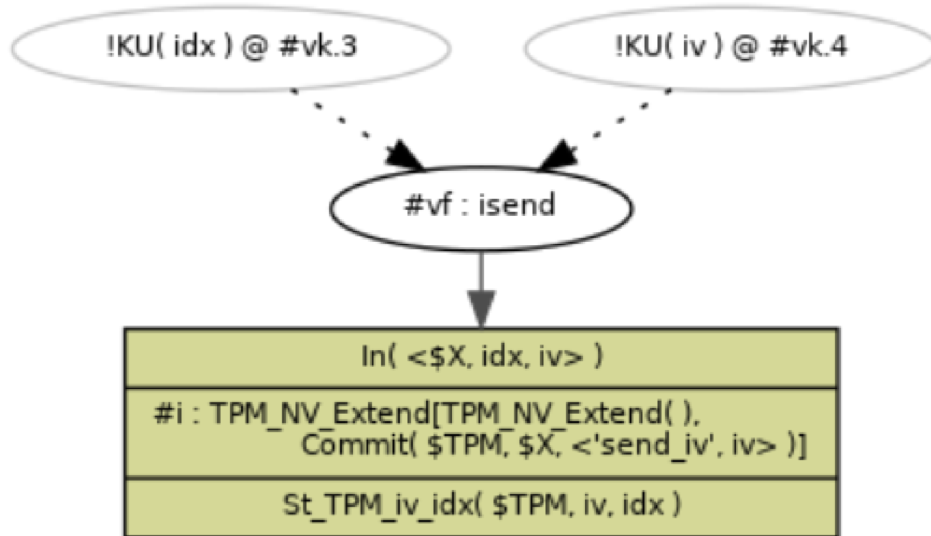

Figure 6.15: Attaching NVPCRs protocol - Aliveness and Injective agreement.

6.2.5 Detaching NVPCRs: Results

The results given by Tamarin for the Detaching NVPCRs protocol model are shown in Figure 6.16 and in Figure 6.17: these two traces represents the ones in which the lemmas are not verified.

The Figure 6.16, shows the trace found for message authentication lemma: this trace found by Tamarin shows how the Attacker can inject any message in the public channel, and in this case it can probably construct the hash of the hash of the configuration policy and its signature and send it to TPM for verification. However, this trace does not represent a possible attack, because the TPM_verifySignature command always verifies the signature.

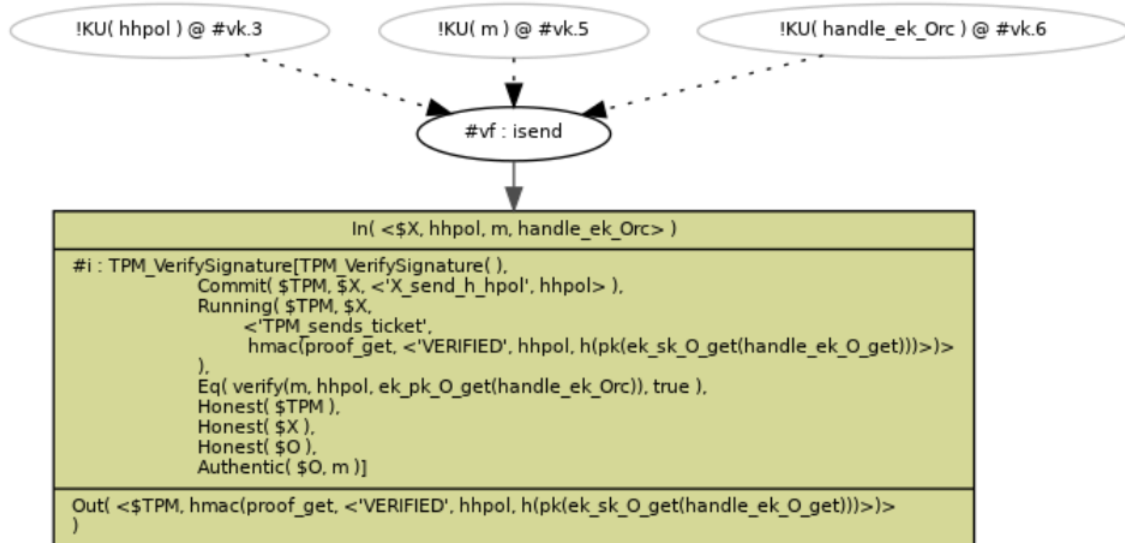

Figure 6.16: Detaching NVPCRs protocol - Message authentication.

The Figure 6.17 shows how the attacker, having full freedom in the public channel, can insert, in this case, all the values that the TPM needs in order to execute the PolicySigned command. This command can be executed without the actual protocol being started and without Orchestrator and X participating. Also in this case, the TPM verifies the values that receives, comparing them to the ones it calculates inside, so this does not represent a possible attack.

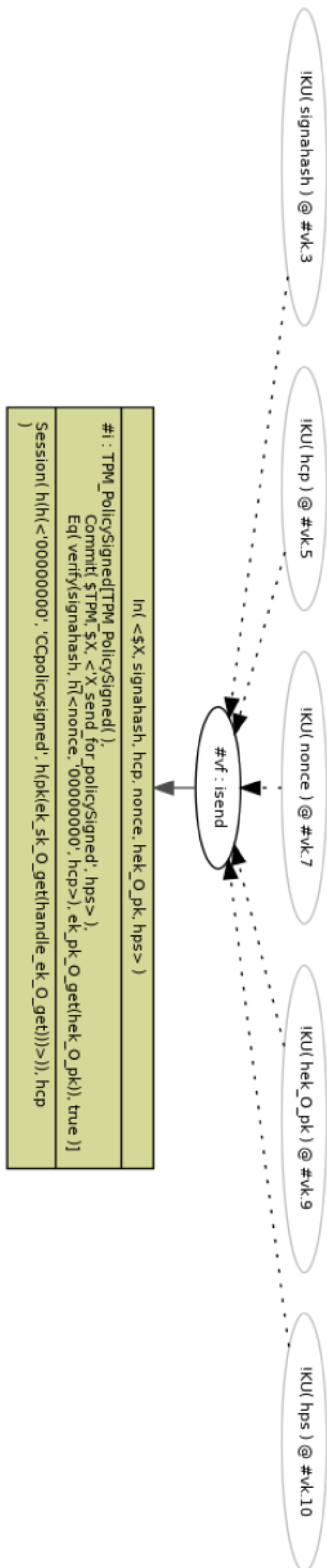

Figure 6.17: Detaching NVPCRs protocol - Aliveness and Injective agreement.

Chapter 7

Conclusions

This chapter will present the conclusion of the thesis work, taking into account the results obtained using the Tamarin prover.

7.1 Conclusion

The thesis document aimed to verify the security level of two kind of remote attestation processes in a fog computing environment:

- Attestation by Quote: remote attestation between the Fog Node and the Orchestrator, based on the Quote;
- Oblivious Remote Attestation: inter-VFs remote attestation based on a challenge;

The Oblivious Remote Attestation process is comprehensive of five protocols:

- Attestation Key creation protocol: protocol managing the creation and provision of the Attestation Key;
- Measurement update protocol: it manages the updates in the configurations;
- ORA protocol: attestation between two nodes;
- Attaching NVPCRs and Detaching NVPCRs protocols: these are the two protocols managing the NV registers' creation and deletion.

Specifically, the protocols analysed have the objective of establish trust between all the involved devices, checking their state correctness in order to know if they have been compromised. The protocols have been symbolically modeled and verified using a specific tool for formal verification, the Tamarin Prover. Following the Tamarin Prover specifications, the protocols have been modelled using a set of rules, describing the changes in the states during the interaction between all the entities involved. The security properties have been modelled using the Lemma feature of Tamarin: lemmas describe the properties to verify using the tool. In particular, a remote attestation protocol should guarantee that some specific security properties are maintained in the execution. The main lemmas describing the security properties analysed in this thesis were:

- Secrecy: it is a property that specifies the need for certain data to not be discoloured, for example symmetric and private asymmetric keys;
- Authentication: provides proof of authenticity of messages and agents interacting during the execution of the protocol;
- Aliveness: it is a form of authentication that guarantees to an agent A aliveness of another agent B if, whenever A completes a run of the protocol, then B has previously been running the protocol;
- Injective Agreement: it is a stronger authentication property that guarantees to an agent A a unique matching partner B in the running of the protocol.

The functional correctness lemma is always verified, for each of the protocols, while the results of the other lemmas, which should be verified for all the traces of the protocols, highlight some possible problems and ways the attacker can interfere in the interactions between entities. In particular, since the protocols have been modelled using a public channel for the communication, the attacker can always insert messages in the channel and send them to real entities, pretending to be the partner in the interaction. These kind of action coming from the attacker can lead to possible DoS attacks and deterioration of system performance. however, apart from DoS attacks, the traces shown by the Tamarin Prover do not represent a compromise of the system. The problems found on the analysed protocols are due to the lack of a peer authentication protocol, that should be used to authenticate the Orchestrator to the platform (X and TPM), and the platform to the Orchestrator, to avoid interference from the attacker and make the agents only handle messages from other authenticated ones. A possible way to implement peer authentication on the channel could be the use of TLS (Transport Layer Security), which is a protocol that can provide communication security over the network. Another solution, designed by Rainbow is the use of DAA, Direct Anonymous Attestation, based on Elliptic-curve cryptography (ECC), which enables remote authentication preserving privacy with anonymity.

7.2 Future implementations

A possible future development of this work could be to try to verify these and others security properties modelling the protocols with another formal verification tool, such as Proverif. In order to avoid authentication problems, the protocols could also be modelled making the entities interact on an authentic channel, instead of a public one, so that the entities interacting always know that the messages are coming from authenticated agents.

Bibliography

[1] Thanassis Giannetsos (DTU) et al. *RAINBOW Attestation Model and Specification*. 2020.

[2] Thanassis Giannetsos (DTU) et al. *RAINBOW Collective Attestation Policy Enablers Design*. 2020.

[3] Thanassis Giannetsos (UBITECH) et al. *RAINBOW Collective Attestation and Runtime Verification*. 2020.

[4] Bruno Blanchet. "Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif". In: *Foundations of Security Analysis and Design VII*. Vol. 8604. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 54–87. ISBN: 9783319100814.

[5] David Challener and Will Arthur. *a Practical Guide To Tpm 2.0 : Using The New Trusted Platform Module in The New Age of Security*. Apress, 2015. ISBN: 9781430265832.

[6] Piergiuseppe Bettassa Copet et al. "Formal verification of LTE-UMTS and LTE–LTE handover procedures". eng. In: *Computer standards and interfaces* 50 (2017), pp. 92–106. ISSN: 0920-5489.

[7] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650.

[8] Dengguo Feng and Tsinghua University Tsinghua University Press. *Trusted Computing: Principles and Applications*. Vol. 2. Advances in computer science. Berlin/Boston: Walter de Gruyter GmbH, 2017. ISBN: 3110476045.

[9] Jonathan Herzog. "A computational interpretation of Dolev–Yao adversaries". In: *Theoretical Computer Science* 340.1 (2005). Theoretical Foundations of Security Analysis and Design II, pp. 57–81. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2005.03.003. URL: https://www.sciencedirect.com/science/article/pii/S0304397505001179.

[10] Antonio Lioy. *Slide del corso di Cybersecurity (01UDROV), Politecnico di Torino: Trusted computing and remote attestation*. 2020.

[11] C. Meadows. "Formal methods for cryptographic protocol analysis: emerging issues and trends". In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 44–54. DOI: 10.1109/JSAC.2002.806125.

[12] J.H. Moore. "Protocol failures in cryptosystems". In: *Proceedings of the IEEE* 76.5 (1988), pp. 594–602. DOI: 10.1109/5.4444.

[13] Ivan Stojmenovic et al. "An overview of Fog computing and its security issues". In: *Concurrency and computation* 28.10 (2016), pp. 2991–3005.

[14] The Rainbow Team. *Rainbow project*. 2020. URL: https://rainbow-h2020.eu/.

[15] The Tamarin Team. *Tamarin-Prover Manual: Security Protocol Analysis in the Symbolic Model*. 2021. URL: https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf.

[16] Gu Tian-yang, Shi Yin-Sheng, and Fang You-yuan. "Research on software security testing". In: *International Journal of Computer and Information Engineering* 4.9 (2010), pp. 1446–1450.

[17] Jorden D. Whitefield. *Formal analysis and applications of direct anonymous attestation.* 2020.