Formal Verification of "The Entity Attestation Token (EAT) draft-ietf-rats-eat-31", BSc proj (Autumn 2024)

Mathias Emil Lystlund Rasmussen - memr@itu.dk

December 16, 2024

Course code: BIBAPRO1PE
IT-University of Copenhagen
Supervisor: Alessandro Bruni

Contents

Abstract

This project focuses on the formal verification of a minimal implementation of the Entity Attestation Token (EAT) protocol, as specified in the draft-ietf-rats-eat-31 [1] and based on the Remote ATtestation procedureS (RATS) achitecture.
The primary goal of the project was to model the protocol using Tamarin-prover and verify its compliance with the security properties: authenticity, integrity and freshness/non-replayability.
Additionally, the protocols level of agreement was assessed using the Hierarchy of Authentication framework described in [2]. The level of post-compromise security was also assessed using the On Post-Compromise security report [3].

The RATS architecture, on which EAT is based, is designed to standardize diverse implementations, leading to specific modeling choices for formal analysis. This report presents a simplified model that assumes a Trusted Execution Environment (TEE) within the attester and combines verifier and relying party roles into one entity. This model also assumes that the verifier and the relying party knows everything in order to verify the evidence send by the attester, thus excluding any need to contact any third party entity.

The formal analysis confirms that the modeled protocol meets its claimed security properties and achieves non-injective agreement from the verifiers perspective. It also reveals that the protocol does not provide authentication or agreement from the attesters perspective.
The protocol achieves weak post-compromise security but lacks a more detailed proof of this.

Acknowledgement

I would like to thank my supervisor, Alessandro Bruni, for his guidance and support during this project. His assistance in understanding Tamarin, remote attestation and identifying useful sources and theories has been crucial in shaping this work. I appreciate the time and feedback he provided throughout the process.

1 Introduction

1.1 Problem Statement

In the realm of cybersecurity, ensuring the integrity and trustworthiness of devices is important. Remote attestation protocols enable one device (the verifier) to determine the state of another device (the attester) remotely.

Despite their importance, protocols often contain subtle errors or vulnerabilities that can be exploited by adversaries. According to the `OWASP Top Ten Project`, cryptographic failures rank as the second most common risk [4]. These failures can lead to severe breaches of security, underlining the necessity of ensuring the robustness and correctness of cryptographic protocols.

To address this problem, formal verification tools like Tamarin Prover are used to analyze the "correctness" of these protocols. Tamarin allows us to create formal models of the protocols and verify that they satisfy their intended security properties.

The goal of this project is to:

- Analyze the proposed architecture described in the `Entity Attestation TOKEN (EAT) draft-ietf-rats-eat-31` [1] and create a formal model reflecting a minimal implementation. The EAT model architecture is based on the `Remote ATtestation procedureS (RATS) Architecture` [5], and the RATS architecture will therefore be an essential element in the modeling of the architecture.
- Outline the claimed security properties and use Tamarin to formally verify that a minimal implementation of the architecture meets the claimed properties.

I will in my analysis of the protocol expand on the term authentication, drawing insights from the report `A Hierarchy of Authentication Specifications` [2]. I will also expand on the adversary model, by including an adversary able to compromise devices as described in the report `On Post-Compromised Security` [3]. This will also lead me to evaluate the models level of Post-Compromised security.

2 The Entity Attestation TOKEN(EAT) draft-ietf-rats-eat-31

2.1 Remote attestation

Remote Attestation (RA) is a security concept/method used to enable a remote peer to verify the state of another system. This system could be any kind of computer device or piece of software etc. The purpose of this attestation would typically be some kind of verification of the integrity and trustworthiness of a remote system.

One scenario (and the scenario we will be looking at throughout this thesis) could be that a system (the attester) wants to get access to some kind of resource.
In order for an attester to be able to be trusted by a relying party, the attester has to share information about itself. This information is vital for the relying party, since it doesn't want to share resources and secret information to just anyone. In order for an attester to be trusted, it has to live up to certain requirements set by the relying party. These can include running a "new enough" software version, not having modified software running, a maintenance log etc.
The purpose of the information send to the relying party is to convince it, that you are in a "good enough state" to share information with.
A reason why a relying party might want to know which software version a device is running could be that there is a known vulnerability in older versions, where adversaries might be able to extract the data send to the attester device [5][6].

The concept of Remote attestation is not new, and is used all over and in many systems. The problem with remote attestation is, that it has traditionally been implemented in a proprietary manner, instead of something that is standardized [5][6].

2.2 Remote ATtestation procedureS (RATS)

As mentioned, the previous problem with attestation was that our systems were proprietary systems for special use cases as opposed to something that was broad and standardized.The `Internet Engineering Task Force (IETF)` took these prior examples of deployed systems of attestation and tried to standardize them.
So RATS is a standardized architecture (not a protocol), which purpose is to create standardized formats within existing/new protocols/systems, in order to achieve mutual attestation [6][5].

The key element in the RATS architecture is that the attester provides evidence about itself and sends it to a verifier. The verifier then evaluates the evidence to what is called Attestation Result. The verifier then sends the attestation result to the relying party, that then decides what access to give the attester [5][6].

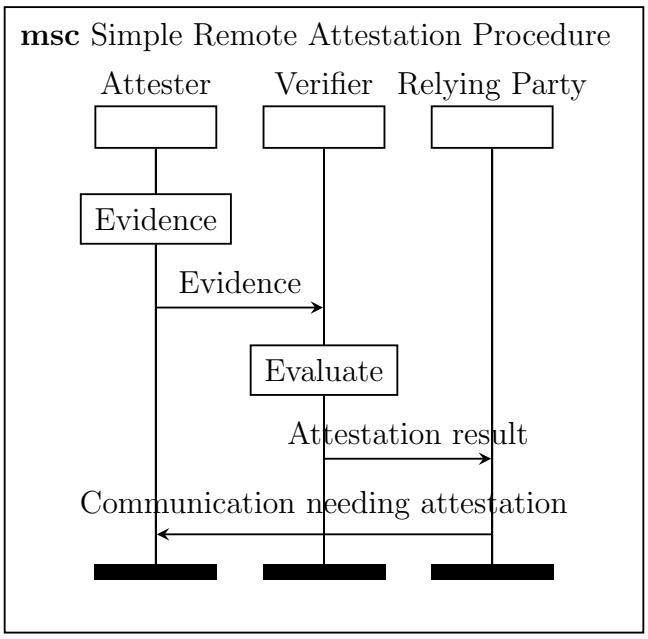

Figure 1: A simplified representation of the Remote Attestation Procedure.

The attester can be any IoT device, computer, server, smart phone etc, trying to gain access to a network, system, or resources and needs to be trusted by the system that it is interacting with.

2.2.1 Evidence

In order for an attester to be able to be trusted by a relying party, the attester has to share information about it self. This information is vital for the relying party, since it doesn't want to share resources and secret information to just anyone. In order for an attester to be trusted, it has to live up to certain requirements set by the relying party. These can include running a "new enough" software version, not having modified software running, a maintenance log etc. The purpose of the information send to the relying party is to convince it, that you are in a "good enough state" to share information with. A reason why a relying party might want to know which software version a device is running could be that there is a known vulnerability in older versions, where adversaries might be able to extract the data sent to the attester device [5][6].
In RATS terminology this is called evidence. As we will see later in this report, the purpose of EAT based attestation is to standardize the evidence.

2.2.2 Trusted Execution Environment (TEE)

If an attester is compromised, an adversary can "lie" about the state of the attester and thereby deceive the relying party and verifier to accept an attester in a bad state. This is the motivation of a TEE.
A TEE provides a separate environment within a device, which that device has very restricted access to. The TEE is a trusted environment that provides trustworthy appraisal for other components. It does so, by collecting, protecting and signing information in the system/IoT device. This information can describe attributes of the

system, such as hardware, software, firmware etc, so in attestation that would be the evidence [5][7].

The TEE's isolation from the main device environment ensures that even if the device itself is compromised, the TEE remains secure and uncompromised, preserving its ability to generate trustworthy evidence.

While a Trusted Execution Environment (TEE) is designed to provide robust security by isolating sensitive operations and protecting critical data, it is not immune to all forms of attacks. Advanced adversaries, particularly those with hardware-level capabilities, can exploit vulnerabilities or weaknesses in a TEE with psychical access to the device [3][7][6].
In RATS [5] they use a general model to model this.

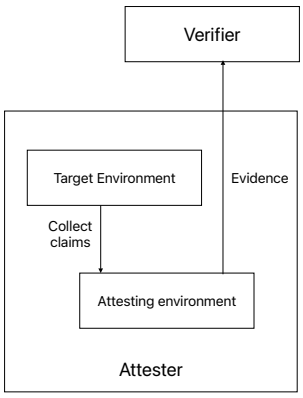

Figure 2: Attester device providing evidence to verifier [5]

Here an attester is modeled as consisting of 2 environments located in the same entity. Target environment and Attesting environment. Claims are collected from the target environment by the Attesting environment. Evidence is then generated and send to the Verifier by the Attesting environment.

I will not delve deeper into the inner workings of a TEE mechanism, as it falls outside the scope of this report and is also beyond the focus of the EAT and RATS specifications. Note that the TEE like mechanism described in the RATS and EAT protocols doesn't have to be implemented with a TEE. There are many other acceptable mechanisms like TEE's that fulfill that role, like TPM, root of trust, Intel SGX, ARM TrustZone etc.

2.2.3 Verifier

The verifiers role in the attestation process, is to evaluate the evidence provided by the attester and determine the trustworthiness of the attesters state. The verifier does this by consulting a Verifier Owner. The Verifier Owner sends what is called an "Appraisal policy for Evidence". This "Appraisal policy for Evidence" defines what evidence is acceptable and how the evidence should be processed, and what endorsements are required [5][6].

The verifier can then contact and receive endorsements from the different endorsers related to the attester. The Endorser can be a manufacturer of different hardware associated with the attester. This endorser can, as an example, verify signatures used by different hardware; like a TEE [5][6].
The verifier can then consult a Reference Value Provider which supplies the verifier with reference values. The reference values are known-good values, which are used to evaluate the evidence from the attester. These "good" reference values can be software updates with no major security faults, hardware configurations, etc.
Following these the verifier will produce an attestation result. The attestation result is an assertion that the attester is in a good state, bad state or maybe its a lot more refined then that.
So the attestation result can be as simple as a boolean; yes, the system is good or no, the system is not good, or it can be a long list of different evaluations of different aspects of the attester [5][6].

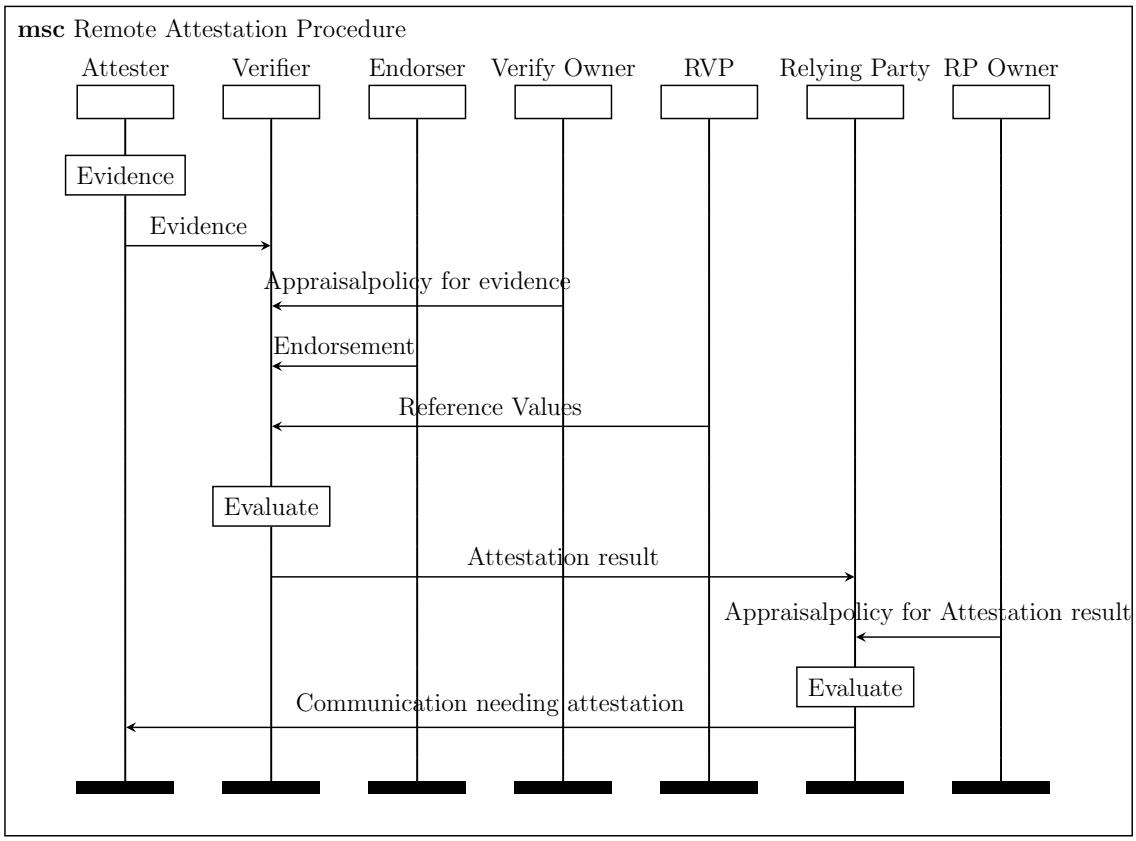

Figure 3: A more in depth depiction of the different steps in a remote attestation procedure. In this figure Reference Value Provider has been reduced to RVP and the Relying Party Owner has been reduced to RP Owner. The boxes with text e.g. the box with evidence, indicates an internal process within that given entity [5][6].

In figure 3 we see an attester, in an internal process, produce evidence and send that evidence to the verifier.
The Verifier then consults an Appraisalpolicy for evidence, Endorser and a Reference Value provider in order to evaluate the evidence. This evaluation is turned into an

attestation result.
The Verifier then sends the attestation result to the relying party that the attester wishes to communicate with/gain resources from. The relying party then consults a relying party owner, which provides an appraisal policy for attestation result. This policy serves as a guideline for the relying party to evaluate whether the attester meets the required security and trust standards before granting access or resources.

Note that the different entities shown in figure 3: Attester, Verifier, Endorser, Verify Owner, etc, are displayed as different entities. This doesn't have to be the case, and in many cases, these entities overlap and are the same device.

2.3 Layered Attestation

In the previous section we stated that an attester needed to have some kind of TEE, to be able to provide trustworthy evidence to the verifier.
A more general model of this is described in the RATS report [5] and is shown in Figure 2.

This model is extendable to a layered attestation model as shown in Figure 4:

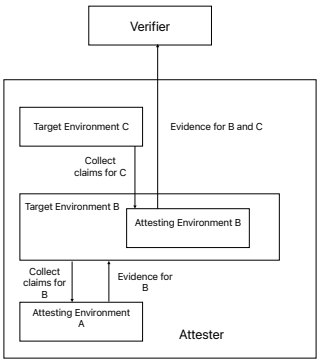

Figure 4: Layered attestation model

In the layered attestation model, you introduce multiple layers of attestation. In this model, a target environment (target environment B in Figure 4) can itself, once it has been evidenced by another attesting environment (target environment A in Figure 4), act as an attesting environment for other target environments (target environment C in Figure 4). It can then produce a combined/chained evidence for the validator.

In the example in Figure 4, A could be some hardware component, B could be some firmware, and C could be an operating system.

2.4 Entity Attestation Token (EAT)

The specific specification of EAT format is The Entity Attestation Token (EAT) draft-ietf[1]-rats-eat-31 [1]. The EAT protocol follows the operational model described in the RATS Architecture [5][1].

EAT is a standardized format used for conveying evidence, in the form of claims about the attester. These claims can include information about the hardware, software, or other attributes essential for the verifier to verify the state of the attester.
The IEFT specification descries a series of possible claims that can be used in an EAT. Which claims are used is very much context dependent - some claims are only relevant for a hardware entity and some are only relevant for a software entity.

The IEFT specification devides claims into 3 groups:

- Claims to ensure freshness e.g. eat_nonce
- Claims describing the entity
- Claims describing the token

An EAT is encoded in either JavaScript Object Notation (JSON) or Concise Binary Object Representation (CBOR), depending on the use case. An EAT is build on JSON Web Token (JWT) and CBOR Web Token (CWT) [1].
The following is an example of a JSON formatted EAT as it could look if a JWT token is used. It is also found in [1]:

```
{
    "eat_nonce": "MIDBNH28iioisjPy",
    "ueid":      "AgAEizrK3Q",
    "oemid":     76543,
    "swname":    "Acme IoT OS",
    "swversion": "3.1.4"
}
```

This payload includes a nonce, which ensures freshness. The Universal Entity ID (ueid) is a "serial number" uniquely identifying the attester device. The Original Equipment Manufacturer ID (oemid) can be used to identify the hardware manufacturer of components in the attester device. The software name (swname) is used to identify the name and version of software running, and the software version (swversion) is the current software version running on the attester [1].

[1]The Internet Engineering Task Force (IETF), is a standards development organization for the Internet[8]

2.5 Json web token (JWT)

JWT is one of the recommented EAT formats in [1]. In the following I will have a short description of JWT and especially how it addresses the security properties.

Json Web Token or JWT is basically a stateless way to communicate some information between 2 (or more) parties in a secure way. Secure here meaning guaranteeing authenticity and Integrity of the content.
It is important to note, that is does NOT guarantee confidentiality - the content/payload is send in plain text (base64 encoded but NOT encrypted).
JWT consists of 3 parts:

- Header
- Payload
- Signature

The figure below shows an example of a JWT containing an EAT as the payload

```
{
    "alg":"HS256",
    "typ":"JWT"
}.
{
    "eat_nonce": "MIDBNH28iioisjPy",
    "ueid":      "AgAEizrK3Q",
    "oemid":     76543,
    "swname":    "Acme IoT OS",
    "swversion": "3.1.4"
}.
5dlp7GmziL2QSO6sZgK4mtaqv0_xX4oFUuTDh1zHK4U
```

The signature is a signed cryptographic hash of the payload, signed with the senders private key.
The signature part is the part that guaranties authenticity and integrity of the content. The signature identifies the sender of the information (authenticity) and the contained hash value can be used to verify, that the information has not been tampered with (Integrity) [9].

2.6 Properties

The Entity Attestation Token protocol should comply with the following properties to its security, assuming that the Trusted Execution Environment (TEE) is not compromised. These properties are interpreted by me using the protocols terminology eg. MUST, MUST NOT, REQUIRES, SHOULD, MAY, MAY NOT, and so on.

- **Authenticity**
  The protocol must ensure that a receiver of an entity attestation token (EAT), is able to verify the authenticity of the EAT. *"An EAT MUST have Authenticity and integrity protection"* [1, p. 11, line 4].

- **Integrity**
  The protocol must guarantee that the EAT has not been tampered with during transmission. An attacker should not be able to modify any claims within an EAT which would be accepted by a verifier. *"An EAT MUST have Authenticity and integrity protection"* [Ibid].

- **Freshness and non-replayability**
  The protocol must ensure that the evidence produced by the attester is fresh, meaning that the data about the attester reflects the current state of the entity. The nonce is used for this, and ensures that an attacker cannot resend a previously valid attestation token. *"Full profiles MUST be complete such that a complying receiver can decode, verify and check for freshness every EAT created by a complying sender"* [1, p. 39, line 20-21].

Note that there are no hard requirement for what an EAT must contain: *"The set of claims that an EAT must contain to be considered valid is context dependent and is outside the scope of this specification"* [1, p. 12, line 17-18]. But an EAT MUST contain a Claims-Set.
The EAT claims that are proposed by the protocol [1, p. 59, line 14-19], do have specifications if they are in an EAT and registered in [IANA.CWT.Claims] and [IANA.JWT.Claims] IANA registries.

The degree of trustworthiness of the claims generated by an attester is beyond the scope of the EAT report [1]. Meaning that mechanisms like TEE's and so on, MAY be implemented, but are not a MUST [1, p. 56-57].

Confidentiality is described as optional in the EAT specification [1, p. 7] and therefore is not required to ensure the proper functioning of the protocol. The minimal implementation of the protocol is designed to operate correctly without it and will therefore not be modeled as a property in my model.

In this project I will model the EAT protocol in the tool Tamarin-prover. Afterwards I will use Tamarin-prover to verify the listed security properties. The tool, Tamarin-prover, will be explained in a later section, in order to understand why a proof created via this tool is useful when assessing the correctness of a protocol.

3 Hierarchy of Authentication

In order to better assess the level of authentication that our protocol fulfills, we evaluate it against the `Hierarchy of Authentication` defined by Gavin Lowe [2]. This framework categorizes authentication into a hierarchy of increasingly strong guarantees. The levels in the hierarchy include aliveness, weak agreement, non-injective agreement, and agreement, each building upon the previous to provide stronger guarantees of authenticity [2].

3.1 Aliveness

A protocol guarantees aliveness to an entity A of another entity B, whenever A (acting as the initiator) completes a run of the protocol, apparently with B, then B has previously been running the protocol with A.
This does not imply that B believed that it was running the protocol with A. This does also not imply that B had been running the protocol "recently" [2].

3.2 Weak agreement

A protocol guarantees weak agreement to an entity A with an entity B if, A completes a run of the protocol, apparently with the responder B. Then B has previously been running the protocol, apparently with A [2].

3.3 Non-injective agreement

A protocol guarantees an entity A non-injective agreement with a responder B on a set of data items if, whenever A completes a run of the protocol, apparently with B, then B has previously been running the protocol, apparently with A, and B was acting as responder in its run, and the two entities agreed on the data items [2].

This does not guarantee a `one-to one` relation between the protocol runs of A and B, meaning A may believe that it has completed two runs of the protocol, while B only has completed one run of the protocol.

3.4 Agreement

A protocol guarantees an entity A agreement with a responder B on a set of data items if, A completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and B was acting as a responder, and the two entities agreed on the data items, and each of the runs of A corresponds to a unique run of B [2].

Properties	Aliveness	Weak Agreement	Non-Injective Agreement	Agreement
Guarantee	The responder was active at some point.	Both entities acknowledge each other's involvement.	Both entities agree on a set of data items, but multiple runs by A may correspond to one run by B.	Both entities agree on a set of data items, and each run by A corresponds to a unique run by B.
Binding to Session	No. The responders response may not be tied to the current session.	No. The interaction may not relate to the same protocol instance.	Yes, but allows multiple mappings (e.g., multiple runs of A to one run of B).	Yes. Ensures one-to-one correspondence between runs of A and B.
Replay Attacks	Replay attacks are possible.	Replay attacks are still possible.	Replay attacks are still possible.	Replay attacks are prevented since interactions are tied to unique protocol instances.
Acknowledgement of Roles	No. It does not confirm the roles of initiator and responder.	Yes. Each party recognizes the other's role.	Yes. Each party recognizes the other's role.	Yes. Both entities recognize their roles in the interaction.
Agree on Data Items	No. There is no agreement on data exchanged.	No. Agreement on data is not guaranteed.	Yes. Both entities agree on shared data items.	Yes. Both entities agree on shared data items for each protocol run.

Table 1: Key Differences in the properties of the different levels of Authentication [2]

4 Post-Compromised Security

In many systems, Trusted Execution Environments (TEE's) are relied upon to ensure the security of cryptographic operations (as described in section: 2.2.2). These TEE's are often assumed to be uncompromised, leaving little consideration for scenarios where hardware attacks trick or even completely compromise the TEE.
In this section we will explore how some modern protocols address these challenges, drawing insight from: OnPost-compromised Security [3], and include their adversarial models to test how our model might fare against them.

The article also introduces a security properties that will be included in the assessment of the security properties of the protocol.

- **Post-Compromise Security (PCS)**

Post-Compromise Security (PCS) is a property that protects future communication after regaining control from a compromised state. Even if an adversary compromises long-term keys or session state, the protocol recover security for future communication.

The two adversarial models that are introduced are weak/partial compromise, and total compromise of the long-term key (i call this full compromise).

Partial compromise assumes that an adversary is able to partially compromise a trusted hardware security module (in our case, the TEE). This means that the adversary gains enough access to the TEE to manipulate its functionality, such as tricking it into signing bad information as if it were valid, but does not fully compromise the TEE or gain access to its long-term key (LTK).

Full compromise is a more severe attack where the adversary learns all the long-term keys of a party. In our case that would be a full compromise of the attester and the TEE.

5 Tamarin-prover

Tamarin prover is a symbolic modeling and analysis tool used for analyzing security protocols [10]. Tamarin takes as input a formally defined protocol, the capabilities of the adversary, and the desired security properties. With these, Tamarin can automatically or manually analyze the security properties of the protocol. If Tamarin terminates and if the properties defined hold, then Tamarin will provide a proof. If Tamarin terminates and the properties don't hold, then Tamarin will provide a counterexample that represents an attack. Tamarin's built-in adversary model is the Dolev-Yao model [11], which assumes a powerful adversary capable of intercepting, modifying, and injecting messages in the network, but is unable to break cryptographic primitives such as encryption or hashing when they are used correctly.

5.1 Rules

The Tamarin language is based on multiset rewrite rules [10].
To analyze a protocol in Tamarin, you have to represent the protocol as a set of rules that model the behavior of the involved entities. These rules describe the input, actions, and outputs of each entity in the protocol.
A rule in Tamarin has three components:

- Premises: Facts or states that must be true before the rule can execute
- Actions Facts: Represents an observable event or action that occurs during the execution of the rule. It is used to track and document the event, and does not have an impact on the protocol itself.
- Conclusion: Describes the result state after the rule is applied

```
rule attester_Gets_Compromised:
    [AttesterState($Attester, 'good_state')]
    --[AttesterCompromise($Attester)]->
    [AttesterState($Attester, 'bad_state')]
```

The rule `attester_Gets_Compromised` models a scenario where an attester transitions from a "good" state to a "bad" state, thus representing a compromise. The rule consists of a premise `[AttesterState( $Attester,'good_state') ]`, which specifies that there must exist an Attester in the state "good_state" for the rule to execute, and a conclusion `[ AttesterState($Attester,'bad_state')]`, that specifies that the new state of the attester after the rules execution is "bad". The action fact `--[AttesterCompromise($Attester)]->`, represents the observable event, that the attester gets compromised, and is used in order to refer to the event in a lemma [10].

5.2 Terms

In Tamarin, messages are represented as terms, which can take the following forms:

- **Constant (c):** Fixed, global, public values.
- **Variables (v):** Represent dynamic components of a protocol, such as random values, public identifiers, or time points.
- **Function applications (f(m1,...,mn):** Messages formed by applying an n-ary function f to other terms (m1,...,mn).

Tamarin distinguishes between four types of variables, marked with specific prefixes:

- **Fresh Variables ($\sim x$):** Used for random values such as keys and nonces. They are generated using a built-in fact `Fr()`, which generates a fresh variable. As seen in the code 9 line 15, where we generate a fresh new random nonce: Fr($\sim n$).
- **Public Variables ($x):** Represents a publicly known value, such as agent identities.
- **Temporal Variables (#x) & (@x) :** These are variables that represent time points in a protocol trace. These appear exclusively in lemmas, and can as an example be used to specify the order of events such as event x happened before event y `(#x<#y)`. The @ is used to link events to a given temporal variable, such as the verifier sends nonce event happened at time i: `VerifierSendsNonce(...)@i`.

Tamarin supports both built-in and user-defined functions. To use these, add the line:

```
builtins: revealing-signing
```

The built-in theory `revealing-signing` provides functions for message revealing, such as `revealSign` used for signing messages, `revealVerify` used for verifying signatures and `getMessage` used to extract a signed message [10] [12].

5.3 Facts

In Tamarin, the state of the system is represented as a multiset of facts. Facts are used in rules to model changes in the protocol, such as sending or receiving messages, generating fresh values, or transitioning the state of an agent.

5.3.1 Linear Facts

Linear facts represent transient state and are consumed when used. If a rule includes a linear fact in its premise and does not include it in its conclusion, then that fact will be removed from the system's state after the rule has been executed. The syntax of a linear fact is starting with an uppercase letter. An example of a linear fact in the code is the nonce created in the `Verifier_send_Nonce_to_Attester` rule: Nonce($\sim n$). The nonce is modeled as a linear fact to ensure it cannot be reused once it has fulfilled its purpose.

For instance, after the verifier sends the nonce to the attester, it expects the same nonce to be returned. Once the verifier verifies that the received nonce matches the one it originally generated, the nonce is no longer needed and is effectively "deprecated." Modeling the nonce as a linear fact ensures that the protocol maintains this behavior and prevents reusing or processing the same nonce multiple times [10] [12].

5.3.2 Persistent Facts

Persistent facts are not consumed when used. These facts remain in the system's state and can be referenced multiple times across different rules. The syntax of a persistent fact is adding an ´!' as prefix to a Fact:

```
!Identity($A, ~ltk, pk(~ltk))
```

5.3.3 Built-In Facts for Communication

Tamarin provides two built-in facts for modeling communication over an untrusted network controlled by the Dolev-Yao adversary.

- **In:** Represents the reception of a message over the network. This appears in the premise of a rule.

```
[In(msg)]
```

- **Out:** Represents the sending of a message over the network. Appears in the conclusion of a rule.

```
[Out(msg)]
```

5.4 Restrictions

Restrictions are logical constraints used to restrict the set of traces to be considered in the protocol analysis [10].

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

This restriction as an example states that for all x and y's to the time i, whenever the fact `Eq(x,y)` holds at time i, then that implies that x and y must be identical. In my protocol, this restriction is used by the verifier to compare values such as nonces, ensuring that the nonce that it sends is the same as it receives [10].

5.5 Lemmas

Lemmas describe the desired security properties of the protocol. In Tamarin, security properties are defined as trace properties, which evaluate the behavior of the protocol over traces of action facts generated during its execution. Tamarin evaluates these lemmas by exploring all possible traces of the protocol, ensuring that the properties hold in every case or identifying counterexamples if they do not.
A trace in Tamarin is a sequence of action facts that represent the events occurring during a protocol execution. Tamarin evaluates lemmas by symbolically exploring all possible traces that the protocol could generate, given its rules and the adversary model. This allows Tamarin to simulate an adversary's actions and test the protocol's security under various attack scenarios.
Tamarin uses two types of lemmas to define security properties: **Universal Lemmas** and **Existence Lemmas**. Universal lemmas specify properties that must hold for all possible traces. An existence lemma or exists-trace lemma, specifies properties that must hold for at least one trace.
If nothing is specified, then a lemma is by default a universal lemma [10].

```
lemma attester_private_key_compromised:
    "
    All Attester Verifier signed_EAT ltk n #i #k .
    (
        Create_indent(Attester, ltk) @k &
        VerifierVerifiesEat(Attester, Verifier,
           signed_EAT, n) @i
    )
    ==> not(Ex #j .
            K(ltk) @j & #j<#i &
            #k<#i)
    "
```

The universal lemma `attester_private_key_compromised` checks that for all attester, verifier, signed_EAT, long-term key (ltk), nonce (n) and time points #i and #k. If an attester creates an identity at time k and a verifier verifies a signed_EAT at time i, then there must not exist an earlier event at time #j where the attesters long-term key was known by the adversary (K(ltk)). This lemma helps to check that for all successful runs of our protocol, that the attesters private key is never compromised.

5.6 Proofs

When Tamarin tries to prove a universal lemma, it does so by negating the lemma and trying to find a trace that satisfies the negated lemma. If tamarin finds a trace, the lemma has been disproved. The trace that Tamarin found is a counterexample and represents an attack that violates the security property expressed by the lemma. On the other hand, if Tamarin shows that there exist no trace for a negated version of the lemma, then Tamarin has proved that the lemma holds. [12][10].

For an existence lemma (exists-trace) Tamarin searches for at least one trace that satisfies the lemma. If such a trace exists, the lemma is considered proven.

The following figure is an example of a trace generated by Tamarin.

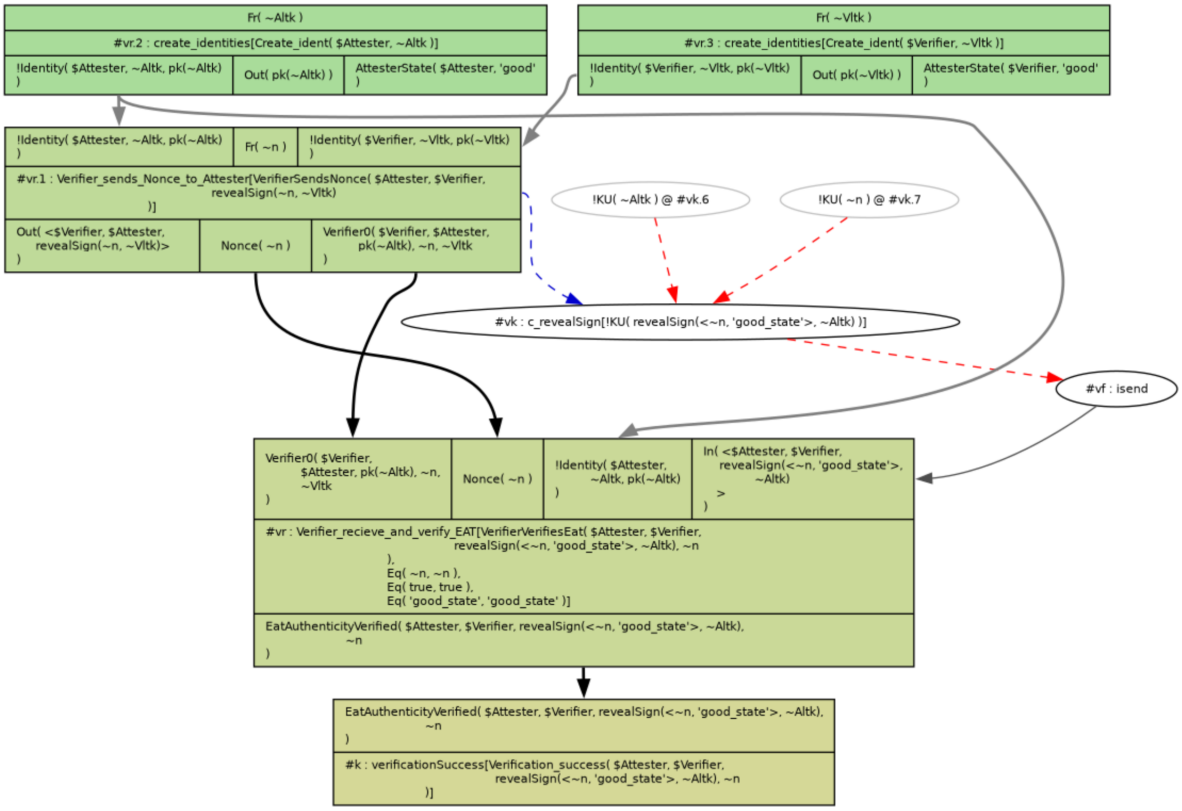

Figure 5: Tamarin trace generated from the sanity check lemma by Tamarin prover 9 line 130.

Each of the green rectangles represents the application of a rule. The top row represents the premise of a rule, the middle row represents the action fact and a time stamp `(ex #vr.1)`. The bottom row represents the conclusion fact.
The black arrows connecting the rules represents linear facts, while the gray arrows represents persistent facts.
The ovals represent the network, and thereby the adversary. The dashed arrows represents actions made by the adversary such as intercepting, modifying, or replaying messages.
The red dashed arrow are used to represent steps taken by the adversary, in this case the adversary learns the nonce `!KU(`$\sim n$`)@#vk.7`.
Dark blue lines: *indicates an ordering constraint deduced from a fresh value: since fresh values are unique, all rule instances using a fresh value must appear after the instance that created the value* [10].

The proof shown in figure 5 can be used to validate that the order of events happen in the correct order, and that none of the "intended" logic of the protocol can be broken.

In this example we also prove that since the EAT send is only encrypted with a private key with a known public key associated with it, that an adversary is able to learn the contents of the message.

6 My model in Tamarin-prover

In this section, I describe my model of the Entity Attestation Token (EAT) protocol that I have implemented in Tamarin-prover. This section will include a more detailed description of the modeled protocol and its properties and my considerations on how to model the protocol, what to include and what to exclude.

The architectures proposed in the EAT [1] and RATS [5] reports, have a problem when it comes to modeling and analyzing them. The problem is "vagueness", meaning that the proposed architectures main purpose is to standardize all the proprietary implementations of attestation, leading to architectures that aren't very specific as to how to implement specific steps of the protocol. This led me to consult sources like intel [13], [7], [6] for clarification on a few of choices you can take, when implementing the EAT/RATS protocol.

The first simplification that I made, was to simplify the protocol, by reducing the amount of actors that it included as seen in figure 3. Here I assumed that entities like endorser, reference values, appraisal policy for attestation result, etc. weren't necessary since a verifier and relying party could in some instances be assumed to know what those entities know, thus they wouldn't have to consult any third party entity for the process of evaluating evidence. The steps of the architecture where entities like endorsers, reference value providers etc.n communicate with the verifier is not specified and I assume that it is done using some form of TLS, thus starting to include these entities, in my view, would take away the focus of the EAT exchange part of the protocol.

I also modeled the verifier and the relying party as the same entity for simplicity and because in some instances they are the same entity [5] [1]. In the code the verifier/relying party entity is called verifier, so from now on if I mention the verifier in this protocol model it is the relying party entity that also has the capability of acting as a verifier.

The modeled protocol structure is displayed in Figure 6.

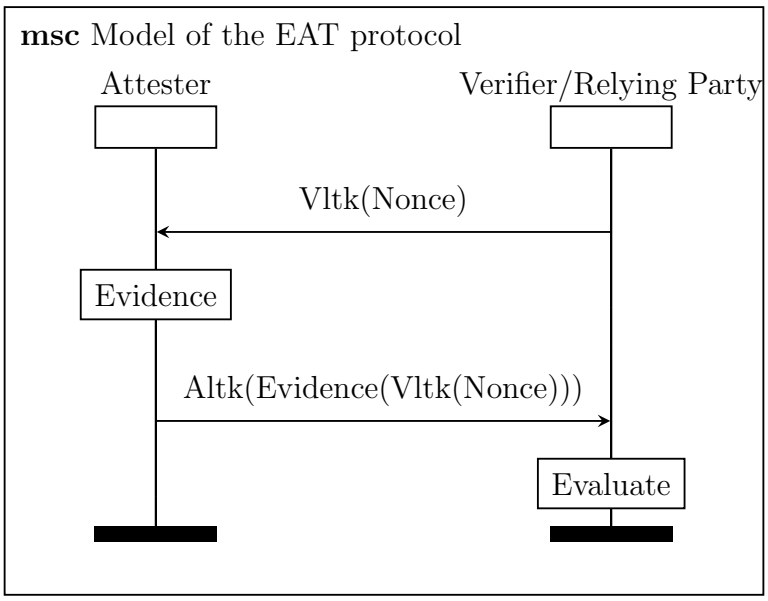

Figure 6: My model in Tamarin only including an attester entity and an entity that is both verifier and relying party

In the model I have implemented a TEE like mechanism. This mechanism is inside the attester and is the one holding and using the cryptographic key. This means that if an adversary compromises the attester it is not able to use the cryptographic key, unless it also compromises or tricks the TEE. A TEE mechanism is described to be outside of the scope of the EAT report, but is still mentioned as a MAY [1].
In the model I use two terms:

- AttesterState
- Attester_data

Attester state represents the actual state that the attester is in. In the model it is used as a form of "meta" state to later verify the actual state an attester was in.
In the model an attester can be in the following states:

- **Good_state:** The attester is not compromised and its data is in a good state.
- **bad_state:** The attester state is bad either because it has bad data or that an adversary has compromised the attester, but is unable to trick or compromise the TEE.
- **Partial_Compromise:** The attester is compromised and the adversary is able to trick the TEE into signing bad attester_data as good attester_data effectively lying to the verifier. Here the adversary does not learn the private key inside the TEE. This is the weak-compromise/partial compromise described in section 4.

While Attester_data refers to the data relayed by the TEE. This does not have to reflect the actual state of the attester. Here attester_data can be either 'good' or 'bad'.

To simplify the model, the contents of the EAT have been reduced to a single attribute representing the attesters EAT information: either `'good'` or `'bad'`. This abstraction allows the model to focus on verifying the protocol's ability to handle correct and incorrect data while reducing complexity. By simulating correctness with a binary state, it is easier to validate the verifiers behavior and identify vulnerabilities in the protocol.
The EAT report specifies that EAT attestation should use either JWT or CWT for ensuring authenticity and integrity. In Tamarin I used the built in function `revealing-signing` and public/private key signing 9. This achieves the same properties and is built in functionality, so it is easier to work with.

6.1 The modeled protocol

6.1.1 Create Identities

The first rule is `Create_identities` and the rule allows the protocol to create n number of entities that all have some name &A, a public/private key pair and an `AttesterState` that indicates if the entity is in a 'good_state', 'bad_state' or 'partially_Compromised' state.

Lastly the rule exposes the public key to the network. This also exposes the public key to the adversary, since the Dolev-Yao model assumes a strong adversary that controls the network.
This rule is used to create both attesters and verifiers.

```
rule Create_identities:
    [Fr(~ltk)]
    --[CreateIdent($A, ~ltk)]->
    [!Identity($A, ~ltk, pk(~ltk)), Out(pk(~ltk)),
       AttesterState($A, 'good')]
```

6.1.2 Verifier sends Nonce to Attester

The next rule `Verifier_sends_Nonce_to_Attester` is also depicted as the first step in Figure 6. In this step the verifier entity creates, encrypts and sends a fresh nonce to the attester.

The EAT specification [1] does not require the nonce to be encrypted. However, I chose to encrypt the nonce in my model for two reasons. First, encryption provides a basic mechanism for the receiver (the attester) to authenticate the nonce, which could prevent certain types of adversarial interference. Second, while the encryption of the nonce is not currently used elsewhere in the model, including it aligns with common practices for ensuring message authenticity in cryptographic protocols.

Note that the nonce is a linear fact, so when the verifier sees the nonce again, the nonce goes out of scope and can no longer be reused by another attester or adversary.

```
rule Verifier_sends_Nonce_to_Attester:
    let
        signed_nonce = revealSign(~n, ~Vltk)
    in
[!Identity($Attester, ~Altk, pk(~Altk)), Fr(~n), !
   Identity($Verifier, ~Vltk, pk(~Vltk))]
--[VerifierSendsNonce($Attester, $Verifier,
   signed_nonce)]->
[Out(<$Verifier, $Attester, signed_nonce>), Nonce(~n),
   Verifier0($Verifier, $Attester, pk(~Altk), ~n, ~Vltk)
   ]
```

6.1.3 Attester_Gets_In_Bad_State and Attester_Gets_Partially_Compromised

The two rules introduce the two adversary models. The `Attester_Gets_In_Bad_State` rule introduces an adversary that can compromise an attester but not trick or compromise the attester's TEE.
While the `Attester_Gets_Partially_Compromised` rule introduces an adversary that can compromise the attester and trick the TEE to sign bad attester_data as good attester_data.

```
rule Attester_Gets_In_Bad_State:
    [AttesterState($Attester, 'good_state')]
    --[AttesterGetsBadState($Attester)]->
    [AttesterState($Attester, 'bad_state')]

rule Attester_Gets_Partially_Compromised:
    [AttesterState($Attester, 'good_state')]
    --[AttesterGetsPartiallyCompromise($Attester)]->
    [AttesterState($Attester, 'partially_Compromised')]
```

6.1.4 Attester_create_and_sends_EAT

The next three rules are `Attester_create_and_sends_EAT_good`, `Attester_create_and_sends_EAT_bad` and `Attester_create_and_sends_EAT_partial_Compromised`. These rules are functionally similar, where they all collect evidence, sign it using the attester's long-term key, and send it to the verifier. The key distinction lies in the state of the attester. This determines if the attester is in a good_state and sends a good EAT, in a bad_state and sends a bad EAT or in a partially_compromised state and sends a good EAT.

```
rule Attester_create_and_sends_EAT_good:
    let
        attester_data = 'good'
        EAT = <signed_nonce, attester_data>
        signed_EAT = revealSign(EAT, ~Altk)
    in
    [
        !Identity($Attester, ~Altk, pk(~Altk)),
        In(<$Verifier, $Attester, signed_nonce>),
        AttesterState($Attester, 'good_state')
    ]
    --[AttesterSendsEatGood($Attester, $Verifier,
       signed_EAT, signed_nonce)]->
    [Out(<$Attester, $Verifier, signed_EAT>)]
```

```
rule Attester_create_and_sends_EAT_bad:
    let
        attester_data = 'bad'
        EAT = <signed_nonce, attester_data>
        signed_EAT = revealSign(EAT, ~Altk)
    in
    [
        !Identity($Attester, ~Altk, pk(~Altk)),
        In(<$Verifier, $Atterster, signed_nonce>),
        AttesterState($Attester, 'bad_state')
    ]
    --[AttesterSendsEatBad($Attester, $Verifier,
       signed_EAT, signed_nonce )]->
    [Out(<$Attester, $Verifier, signed_EAT>)]


rule Attester_create_and_sends_EAT_partial_Compromised:
    let
        attester_data = 'good'
        EAT = <signed_nonce, attester_data>
        signed_EAT = revealSign(EAT, ~Altk)
    in
    [
        !Identity($Attester, ~Altk, pk(~Altk)),
        In(<$Verifier, $Attester, signed_nonce>),
        AttesterState($Attester, 'partially_Compromised
           ')
    ]
    --[AttesterSendsEatPartialCompromise($Attester,
       $Verifier, signed_EAT, signed_nonce)]->
    [Out(<$Attester, $Verifier, signed_EAT>)]
```

6.1.5 Verifier_receive_and_verify_EAT

The last two rules are `Verifier_receive_and_verify_EAT` and `verification_Success`, where the verification success rule is to underline in the tamarin output, that the verification of the EAT successfully completes and could have been removed. The `Verifier_receive_and_verify_EAT` rule verifies the signature using the attester's public key, thus validating authenticity and integrity. It then compares the nonce in the EAT, to the nonce it sent out to the attester, in order to ensure that the EAT received isn't a replay. Lastly it checks that the EAT information is good. If all of these steps succeeds, then the protocol successfully sends a verification success. Note that a verifier in this model is unable to detect a partial compromised attester.

```
rule Verifier_receive_and_verify_EAT:
    let
        EAT = getMessage(signed_EAT)
        check_signature = revealVerify(signed_EAT, EAT,
            pk(~Altk))
        unpacked_nonce = fst(EAT)
        unpacked_state = snd(EAT)
    in
[Verifier0($Verifier, $Attester, pk(~Altk), ~n, ~Vltk),
    Nonce(~n), !Identity($Attester, ~Altk, pk(~Altk)),
   In(<$Attester, $Verifier, signed_EAT>)]
--[VerifierVerifiesEat($Attester, $Verifier, signed_EAT
   , ~n),
    Eq(unpacked_nonce, ~n),
    Eq(check_signature, true),
    Eq(unpacked_state, 'good')
]->
[...]
```

Figure 7: Verifier verifies the EAT rule

6.2 Properties

6.2.1 Sanity check

The `Sanity_check` lemma's purpose, is to confirm that the protocol's logical flow operates as intended.

The lemma loosely states that if the protocol successfully completes with a `VerificationSuccess` event, then that implies that either:

- There exists an event where the verifier sends a nonce, followed by an event `AttesterSendsEATGood` where the attester uses that nonce to send an EAT in a good state. Note that the `AttesterSendsEATGood` displays Verifier1 and not verifier. Verifier is the entity that initiates the protocol, while verifier1 is some verifier not necessarily the same as the initiator. This helps Tamarin account for traces where another verifier might end the protocol, which is unintended behavior in our model.
- Alternatively, the attester was partially compromised, allowing the adversary to trick the attester's TEE and deceive the verifier.

```
lemma sanity_check:
"
All Attester Verifier signed_EAT n #k .
    (
        VerificationSuccess(Attester, Verifier, signed_EAT,
            n) @k
    )
    ==>
    (
        (Ex Verifier1 vltk #t #i .
           VerifierSendsNonce(Attester, Verifier,
              revealSign(n, vltk)) @t &
           AttesterSendsEatGood(Attester, Verifier1,
              signed_EAT, n) @i &
            #t < #k &
            #i < #k) |
            (Ex #j. AttesterGetsPartiallyCompromise(
               Attester) @j & #j < #k)

    )
"
```

6.2.2 Cannot Verify A Bad EAT or Compromised Attester

The `cannot_Verify_A_Bad_EAT_or_Compromised_Attester` lemma states that for any successful completion of the protocol it implies that there was a `VerifierSendsNonce` event followed by an `AttesterSendsEATGood` event, and there did not happen an `AttesterGetsBadState` before the `AttesterSendsEATGood` event. OR the attester was partially compromised.

```
lemma cannot_Verify_A_Bad_EAT_or_Compromised_Attester:
"
All Attester Verifier signed_EAT n #k .
    (
        VerificationSuccess(Attester, Verifier, signed_EAT,
            n) @k
    )
    ==>
    (
        ((Ex vltk Verifier1 #t #i .
            VerifierSendsNonce(Attester, Verifier,
               revealSign(n, vltk)) @t &
            AttesterSendsEatGood(Attester, Verifier1,
               signed_EAT, n) @i &
            #t < #k &
            #t < #i &
            #i < #k &
            (not (Ex #j. AttesterGetsBadState(Attester) @j
               & #j < #i)))
        ) |
        (Ex #j. AttesterGetsPartiallyCompromise(Attester)
           @j & #j < #k)
    )
"
```

For a `VerificationSuccess` event to occur, a `VerifierVerifiesEat` event has to have successfully completed.
A `VerifierVerifiesEat` event successfully completes, only if the following properties are met:

- **Authenticity of the EAT:** The verifier only accepts an EAT that has been signed by the correct attester using its long-term private key. As described in section 2.6 on the authenticity property.
- **Integrity of the EAT:** By successfully verifying the signature of the EAT using the attester's public key, assuming that no one besides the attester knows the attester's long-term private key, then no one can tamper with the contents of the EAT, doing transmission, without it being noticed. As described in section 2.6 on the integrity property.

- **Nonce freshness:** By successfully comparing the sent nonce to the received nonce, the verifier ensures that no one can reuse EAT data previously accepted. As described in section 2.6 on the freshness and non-replayability property.

6.2.3 attester_private_key_compromised and verifier_private_key_compromised

The two lemmas `attester_private_key_compromised` and `verifier_private_key_compromised` are two lemmas with the same purpose to verify that doing the protocols execution, the adversary is not able to learn the two entities long-term private keys.

```
lemma attester_private_key_compromised:
 "
 All Attester Verifier signed_EAT ltk n #i #k .
    (
        CreateIdent(Attester, ltk) @k &
        VerifierVerifiesEat(Attester, Verifier, signed_EAT,
            n) @i
    )
    ==> not(Ex #j .
                K(ltk) @j & #j<#i &
                #k<#i)
 "
```

The `verifier_private_key_compromised` lemma can be seen in 9.

The privacy of the attester's private key is a prerequisite for all the security properties in the model. Neither integrity, authenticity nor freshness can be guaranteed if the attester's private key has been exposed.
Another reason for including this lemma, is for a future expansion on the adversary model, where the adversary would be able to fully compromise [3] the attester and its TEE, outputting the private key.

6.2.4 nonce_freshness_across_sessions

The `nonce_freshness_across_sessions` might be a trivial lemma, but it ensures that if an event `VerifierSendNonce` sends a nonce n to a time i, then if another `VerifierSendNonce` happens at time j that sends the exact same nonce. Then the two events are the same:

```
lemma nonce_freshness_across_sessions:
 "
 All Attester Verifier n #i #j .
     (
         VerifierSendsNonce(Attester, Verifier, n) @i &
         VerifierSendsNonce(Attester, Verifier, n) @j
     )
     ==> #i = #j
 "
```

6.2.5 attester_does_not_agree_on_nonce_origin

This lemma shows that agreement doesn't hold from the attester's point of view. It shows that if an attester responds to receiving a nonce by the event `AttesterSendsEatGood`, then there doesn't have to exist an event where a verifier sends a nonce `VerifierSendsNonce`. This is because there is no verification of the nonce specified in the EAT or RATS protocols [1] [5].

```
lemma attester_does_not_agree_on_nonce_origin:
    exists-trace
    "
    All Attester Verifier signed_EAT n #t .
        (
            AttesterSendsEatGood(Attester, Verifier,
               signed_EAT, n) @t
        )
        ==>
            not (Ex vltk #j .
                VerifierSendsNonce(Attester, Verifier,
                   revealSign(n, vltk)) @j &
                #j < #t
        )
    "
```

6.2.6 adversary_learns_the_EAT_information

Since the protocol doesn't require confidentiality, that means that a property of the protocol is that an adversary is able to learn both the nonce and the rest of the EAT information sent by both the verifier and attester.
As described in the 2.6 section, confidentiality is an optional property.

```
lemma adversary_learns_the_EAT_information:
 "
 All Attester Verifier signed_EAT n #i .
     (
        VerificationSuccess(Attester, Verifier, signed_EAT,
            n) @i
     )
     ==> (
             (Ex verifier1 EAT Altk #j .
                AttesterSendsEatGood(Attester, verifier1,
                revealSign(EAT, Altk), n) @j) |
             (Ex verifier1 EAT Altk #j .
                AttesterSendsEatPartialCompromise(Attester,
                verifier1, revealSign(EAT, Altk), n) @j)
          ) &
        ((Ex EAT #k . K(EAT) @k) & (Ex n #k1 . K(n) @k1))
 "
```

7 Findings

The goal of this project was to model a minimal version of the Entity Attestation Token (EAT) protocol, as outlined in the EAT [1] and RATS [5] specifications, and validate its compliance with key security properties, namely authenticity, integrity, and non-replayability, as defined in Section 2.6.
The results of the formal verification using Tamarin-prover confirmed that the protocol fulfills these properties, but also revealed certain nuances and limitations in its design.

7.1 Verification of Security Properties

7.1.1 Authenticity

The property of authenticity, as described in section 2.6, ensures that the verifier can verify the authenticity of the received Entity Attestation Token (EAT).
This was confirmed by the lemma `cannot_Verify_A_Bad_EAT_or_Compromised_Attester`, which demonstrates that:

- The verifier only accepts EATs signed with the intended attesters long-term private key, which remains uncompromised during protocol execution (validated by the `attester_private_key_compromised` lemma)

7.1.2 Integrity

Integrity ensures that the content of the EAT has not been tampered with during transmission. This property was also validated by the `cannot_Verify_A_Bad_EAT_or_Compromised_Attester` lemma, as the verifier checks the signature of the EAT. Any modification to the EAT's contents would result in a signature mismatch, ensuring that only unaltered tokens are accepted.

7.1.3 Freshness and Non-Replayability

Freshness and Non-Replayability ensures that an attacker cannot reuse a previously valid EAT to fool the verifier. This property is achieved through the use of a fresh nonce, verified in the `Verifier_receive_and_verify_EAT` rule. By modeling the nonce as a linear fact, the protocol ensures that the same nonce cannot be reused across different sessions.
The `nonce_freshness_across_sessions` lemma additionally validates this property by confirming that each nonce is unique to a specific session.

7.2 Partial Authentication and Agreement Levels

While the protocol ensures authenticity, integrity, and freshness/non-replayability for the verifier, it does not provide any authentication from the attester's perspective. Specifically, the attester does not authenticate the source of the nonce. This is reflected in the `attester_does_not_agree_on_nonce_origin` lemma, which demonstrates that an attester can participate in the protocol with an adversary rather than the intended verifier.
Consequently, the protocol does not establish any level of agreement from the attester's perspective.

Using the Hierarchy of Authentication [2], we can assess the protocol's level of agreement from the perspective of the verifiers:

- **Aliveness:** The protocol guarantees aliveness since the attester must be active to generate and send the EAT. The verifier can infer that the attester participated in the protocol by the signature used in the EAT.
- **Weak Agreement:** The protocol meets weak agreement, as the attester acknowledges the verifiers involvement (by responding to the nonce), and the verifier processes the received EAT.
- **Non-injective Agreement:** The protocol achieves non-injective agreement since the verifier and attester agree on the exchanged data (the nonce and EAT data).
- **Agreement:** The protocol does not fully achieve agreement, since there isn't a one-to-one relation. This is due to the lack of nonce authentication by the attester.

7.3 Confidentiality and Adversary observations

The EAT specification describes confidentiality as optional [1]. Consequently, the minimal model does not implement confidentiality as a security property. This decision is validated by the `adversary_learns_the_EAT_information` lemma, which confirms that an adversary can access both the nonce and the EAT data.

7.4 Insights on Post-Compromised security

Tamarin showed via the lemma `cannot_Verify_A_Bad_EAT_or_Compromised_Attester`, that an adversary's partial compromise of the attester device could lead to a verifier successfully verifying an attester in a bad state. This violates the main purpose of the RATS protocol, that is verifying the state of the attester.

While the following claim has not been formalized in a lemma and therefore remains unproven, I would argue that if an attester regains control of the device, the attester would be able to securely execute the protocol without needing to change its long-term key. The protocol therefore would achieve what I would call weak post-compromise security.

Post-compromise security as defined in [3] means that a system is able to establish a secure session after a full compromise including exposure of the long-term key.
Therefore the protocol only achieves a weaker form of post-compromise security.

8 Discussion

The primary purpose of the RATS architecture is to standardize the diverse and proprietary implementations of remote attestation systems. To achieve this, the architecture is intentionally designed to be general and flexible, accommodating a wide range of solutions for different steps. This introduces challenges for a formal modeling and verification of the architecture.

As a modeler of the architecture, I was required to make specific decisions about how to implement various steps within the protocol.
These decisions include a TEE like mechanism, combining verifier and relying party into one entity, assume that both verifier and relying party doesn't need to consult any third party entity for validating evidence, and represent evidence as a binary state.
These choices shaped my model, resulting in a proof that is highly specific to the particular implementation I modeled. This led to a formal proof that is not applicable to all potential implementations of the RATS architecture. This instead proves that the specific interpretation of the architecture meets the tested properties.

The primary objective of EAT is to standardize the format of evidence, and it is largely based on the RATS architecture. The focus of my formal proof has therefore largely been on the RATS architecture itself rather than the specifications of EAT.
The verification process therefore primarily addressed the architectural principles and mechanisms of RATS. As a result, the findings and proofs presented here are more reflective of RATS' properties and assumptions than of any specific features unique to EAT.

9 Conclusion

The goal of this project was to analyze and formally verify a minimal implementation of the `Entity Attestation Token (EAT) draft-ietf-rats-eat-31` protocol, as described in the EAT [1] and RATS [5] specifications.
The goal was also to outline the claimed security properties that a minimal implementation of the protocols would have. To achieve this, the protocol and its properties were modeled using the tool Tamarin-prover. Tamarin allowed for a formal analysis of the protocols compliance with its claimed security properties. The security properties included authenticity, integrity and freshness/non-replayability.
The analysis also included the theory described in the report
`A Hierarchy of Authentication Specification` [2], which was used in order to assess the level of agreement that the protocol achieved. Further the theory described in the report `On Post-compromised Security` [3] was used to assess the protocols post-compromise security.

Through the formal modeling and verification process, the protocol was shown to meet the following properties:

- **Authenticity:** The protocol must ensure that a receiver of an entity attestation token (EAT), is able to verify the authenticity of the EAT.
- **Integrity:** The protocol must guarantee that the EAT has not been tampered with during transmission. An attacker should not be able to modify any claims within an EAT which would be accepted by a verifier.
- **Freshness and Non-Replayability:** The protocol must ensure that the evidence produced by the attester is fresh, meaning that the data about the attester reflects the current state of the entity. The nonce is used for this, and ensures that an attacker cannot resend a previously valid attestation token.

My formal verification of the minimal EAT protocol concludes that the protocol achieves its primary security properties.

The analysis showed that the protocol achieves a non-injective level of agreement from the verifier's side, and no level of agreement from the attester's side.

The analysis further showed that the protocol achieves a weak post-compromise security against an adversary able to partially compromise the attester device and its TEE.

References

[1] Lundblade Laurence, Mandyam Giridhar, O'Donoghue Jeremy, and Wallace Carl. The entity attestation token (eat) draft-ietf-rats-eat-31, 2024. URL: https://datatracker.ietf.org/doc/draft-ietf-rats-eat/.

[2] Gavin Lowe. A hierarchy of authentication specifications, 1997. URL: https://conferences.computer.org/sp/pdfs/csf/1997/1997-lowe-hierarchy.pdf.

[3] Cohn-Gordon Katriel, Cremers Cas, and Garratt Luke. On post-compromised security, 2016. URL: https://ieeexplore.ieee.org/document/7536374.

[4] OWASP Foundation. Owasp top ten, 2024. URL: https://owasp.org/www-project-top-ten/.

[5] Birkholz Henk, Thaler Dave, Richardson Michael, Smith Ned, and Pan Wei. Remote attestation procedures (rats) architecture, 2023. URL: https://datatracker.ietf.org/doc/rfc9334/.

[6] Confidential Computing Consortium. Remote attestation procedures (rats) architecture, 2021. URL: https://www.youtube.com/watch?v=j30PhEWKgG4.

[7] Confidential Computing Consortium. Tpm based attestation - how can we use it for good?" - matthew garrett (lca 2020), 2020. URL: https://www.youtube.com/watch?v=FobfM9S9xSI&ab_channel=linux.conf.au.

[8] IETF. Introduction to the ietf, 2024. URL: https://www.ietf.org/about/introduction/.

[9] Mostafa Ibrahim. What is a jwt? understanding json web tokens, 2024. URL: https://supertokens.com/blog/what-is-jwt.

[10] The Tamarin Team. Tamarin-prover manual, 2016. URL: https://tamarin-prover.com/manual/master/tex/tamarin-manual.pdf.

[11] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, pages 198–208, 1983. URL: https://ieeexplore.ieee.org/document/1056650.

[12] Xenia Hofmeier. Formal analysis of web single-sign on protocols using tamarin, 2019. URL: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/ba-19-hofmeier-oidc.pdf.

[13] Intel. Intel® trust authorityn. URL: https://docs.trustauthority.intel.com/main/articles/concept-patterns.html?tabs=passport.

Appendix: Tamarin Code Base

The following is the complete Tamarin code base used in this project:

```
theory ThirdEATImplementation
begin
builtins: revealing-signing

rule create_identities:
    [Fr(~ltk)]
    --[CreateIdent($A, ~ltk)]->
    [!Identity($A, ~ltk, pk(~ltk)), Out(pk(~ltk)), AttesterState
       ($A, 'good_state')]


rule Verifier_sends_Nonce_to_Attester:
    let
        signed_nonce = revealSign(~n, ~Vltk)
    in
    [!Identity($Attester, ~Altk, pk(~Altk)), Fr(~n), !Identity(
       $Verifier, ~Vltk, pk(~Vltk))]
    --[VerifierSendsNonce($Attester, $Verifier, signed_nonce)]->
    [Out(<$Verifier, $Attester, signed_nonce>), Nonce(~n),
       Verifier0($Verifier, $Attester, pk(~Altk), ~n, ~Vltk)]


// rule attester gets compromised or has bad EAT data
rule Attester_Gets_In_Bad_State:
    [AttesterState($Attester, 'good_state')]
    --[AttesterGetsBadState($Attester)]->
    [AttesterState($Attester, 'bad_state')]

// Rule for an attester to get partially compromised -> meaning
   that an
// adversary compromises the attester and is able to make the
   TPM sign of information as if it was in a good state.
rule Attester_Gets_Partially_Compromised:
    [AttesterState($Attester, 'good_state')]
    --[AttesterGetsPartiallyCompromise($Attester)]->
    [AttesterState($Attester, 'partially_Compromised')]


// AttesterState($Attester, 'good') -[CompleteCompromise(
   $Attester)]-> Out(~Altk)
// AttesterState($Attester, 'bad') -[RegainControl($Attester)]->
    AttesterState($Attester, 'good')


//============================================================

```

```
40
41 rule Attester_create_and_sends_EAT_good:
42     let
43         attester_data = 'good'
44         EAT = <signed_nonce, attester_data>
45         signed_EAT = revealSign(EAT, ~Altk)
46     in
47     [
48         !Identity($Attester, ~Altk, pk(~Altk)),
49         In(<$Verifier, $Attester, signed_nonce>),
50         AttesterState($Attester, 'good_state')
51     ]
52     --[AttesterSendsEatGood($Attester, $Verifier, signed_EAT,
          signed_nonce)]->
53     [Out(<$Attester, $Verifier, signed_EAT>)]
54
55 // rule for offering a signing interface to the attacker when
      the attester is compromised
56 rule Attester_create_and_sends_EAT_bad:
57     let
58         attester_data = 'bad'
59         EAT = <signed_nonce, attester_data>
60         signed_EAT = revealSign(EAT, ~Altk)
61         // there's no explicit check for the authenticity and
              integrity of signed_nonce
62     in
63     [
64         !Identity($Attester, ~Altk, pk(~Altk)),
65         In(<$Verifier, $Atterster, signed_nonce>),
66         AttesterState($Attester, 'bad_state')
67     ]
68     --[AttesterSendsEatBad($Attester, $Verifier, signed_EAT,
          signed_nonce )]->
69     [Out(<$Attester, $Verifier, signed_EAT>)]
70
71 rule Attester_create_and_sends_EAT_partial_Compromised:
72     let
73         attester_data = 'good'
74         EAT = <signed_nonce, attester_data>
75         signed_EAT = revealSign(EAT, ~Altk)
76     in
77     [
78         !Identity($Attester, ~Altk, pk(~Altk)),
79         In(<$Verifier, $Attester, signed_nonce>),
80         AttesterState($Attester, 'partially_Compromised')
81     ]
82     --[AttesterSendsEatPartialCompromise($Attester, $Verifier,
          signed_EAT, signed_nonce)]->
83     [Out(<$Attester, $Verifier, signed_EAT>)]
84
```

```
//============================================================



rule Verifier_receive_and_verify_EAT:
    let
        EAT = getMessage(signed_EAT) // Get message without key,
            only something a "simulator actor" can do
        check_signature = revealVerify(signed_EAT, EAT, pk(~Altk
           ))

        unpacked_nonce = fst(EAT)
        unpacked_state = snd(EAT)
    in
    [Verifier0($Verifier, $Attester, pk(~Altk), ~n, ~Vltk),
       Nonce(~n), !Identity($Attester, ~Altk, pk(~Altk)),  In(<
       $Attester, $Verifier, signed_EAT>)]
    --[VerifierVerifiesEat($Attester, $Verifier, signed_EAT, ~n)
       ,
        Eq(unpacked_nonce, ~n),
        Eq(check_signature, true),
        Eq(unpacked_state, 'good')
      ]->
    [EatAuthenticityVerified($Attester, $Verifier, signed_EAT, ~
       n)]



rule verification_Success:
    [EatAuthenticityVerified($Attester, $Verifier, signed_EAT, ~
       n)]
    --[VerificationSuccess($Attester, $Verifier, signed_EAT, ~n)
       ]->
    []


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


restriction OnlyOneIdentity:
    "All A ltk1 ltk2 #i #j. CreateIdent(A,ltk1) @i & CreateIdent
       (A,ltk2) @j ==> ltk1=ltk2"



//============================================================
// Lemmas
//============================================================
```

```
126
127
128
129 // sanity check: the protocol gets to the end (i.e. we reach the
        event Verification_success)
130 //even if the AttesterGetsPartiallyCompromise event happens
131 lemma sanity_check:
132 "
133 All Attester Verifier signed_EAT n #k .
134     (
135         VerificationSuccess(Attester, Verifier, signed_EAT, n)
              @k
136     )
137     ==>
138     (
139         (Ex Verifier1 vltk #t #i .
140            VerifierSendsNonce(Attester, Verifier, revealSign(n,
                 vltk)) @t &
141            AttesterSendsEatGood(Attester, Verifier1, signed_EAT,
                 n) @i &
142             #t < #k &
143             #i < #k) |
144             (Ex #j. AttesterGetsPartiallyCompromise(Attester) @j
                  & #j < #k)
145
146     )
147 "
148
149 //The verifier can't successfully verify an attester that is in
       a "bad state".
150 //The verifier can't successfully verify an eat with a wrong
       nonce
151 lemma cannot_Verify_A_Bad_EAT_or_Compromised_Attester:
152 "
153 All Attester Verifier signed_EAT n #k .
154     (
155         VerificationSuccess(Attester, Verifier, signed_EAT, n)
              @k
156     )
157     ==>
158     (
159         ((Ex vltk Verifier1 #t #i .
160             VerifierSendsNonce(Attester, Verifier, revealSign(n,
                  vltk)) @t &
161             AttesterSendsEatGood(Attester, Verifier1, signed_EAT
                 , n) @i &
162             #t < #k &
163             #t < #i &
164             #i < #k &
165             (not (Ex #j. AttesterGetsBadState(Attester) @j & #j
```

```
                < #i)))
        ) |
        (Ex #j. AttesterGetsPartiallyCompromise(Attester) @j & #
            j < #k)
    )
"



//K never learns the attesters long term key
lemma attester_private_key_compromised:
 "
 All Attester Verifier signed_EAT ltk n #i #k .
    (
        CreateIdent(Attester, ltk) @k &
        VerifierVerifiesEat(Attester, Verifier, signed_EAT, n)
            @i
    )
    ==> not(Ex #j .
                K(ltk) @j & #j<#i &
                #k<#i)
 "

 //K never learns the verifiers long term key
lemma verifier_private_key_compromised:
 "
 All Attester Verifier signed_EAT Altk Vltk n #c1 #c2 #i .
    (
        CreateIdent(Attester, Altk) @c1 &
        CreateIdent(Verifier, Vltk) @c2 &
        VerifierVerifiesEat(Attester, Verifier, signed_EAT, n)
            @i
    )
    ==> not(Ex #j .
                K(Vltk) @j & #j<#i &
                ((#c1<#i)&(#c2<#i))
             )
 "


lemma nonce_freshness_across_sessions:
 "
 All Attester Verifier n #i #j .
     (
        VerifierSendsNonce(Attester, Verifier, n) @i &
        VerifierSendsNonce(Attester, Verifier, n) @j
     )
     ==> #i = #j
 "
```

```
212
213 // add a lemma that shows that agreement doesn't hold from the
       attester's point of view
214 lemma attester_does_not_agree_on_nonce_origin:
215 exists-trace
216 "
217 All Attester Verifier signed_EAT n #t .
218     (
219         AttesterSendsEatGood(Attester, Verifier, signed_EAT, n)
               @t
220     )
221     ==>
222         not (Ex vltk #j .
223             VerifierSendsNonce(Attester, Verifier, revealSign(n,
                   vltk)) @j &
224             #j < #t
225     )
226 "
227
228
229 //Lemma shows that an adversary learns the values in the EAT
230 lemma adversary_learns_the_EAT_information:
231  "
232  All Attester Verifier signed_EAT n #i .
233      (
234         VerificationSuccess(Attester, Verifier, signed_EAT, n)
               @i
235      )
236      ==> (
237             (Ex verifier1 EAT Altk #j . AttesterSendsEatGood(
                   Attester, verifier1, revealSign(EAT, Altk), n) @j
                   ) |
238             (Ex verifier1 EAT Altk #j .
                   AttesterSendsEatPartialCompromise(Attester,
                   verifier1, revealSign(EAT, Altk), n) @j)
239          ) &
240         ((Ex EAT #k . K(EAT) @k) & (Ex n #k1 . K(n) @k1))
241  "
242
243
244 end
```