A Detailed Formal Security Analysis of the Protocols Introduced in "Authentication Protocols for Drone Remote Identification" Using the Tamarin Prover

This document assumes that the reader has read our main paper **("Authentication Protocols for Drone Remote Identification")** and possesses introductory knowledge of the Tamarin Prover tool.

1 Introduction

To formally verify the correctness of our proposed protocol, we employed the Tamarin Prover, a symbolic formal verification tool specifically designed for the analysis of security protocols. Tamarin allows protocol designers to rigorously reason about security properties by expressing them as logical lemmata over possible protocol traces.
Tamarin operates under the well-established **Dolev-Yao adversary model**, which assumes a fully compromised communication channel. In this model, the adversary has complete control over the network: they can intercept, delete, replay, modify, and inject messages at will. However, their capabilities are constrained by the principles of symbolic cryptography. That is, adversaries are treated as computationally unbounded but cryptographically limited: they cannot forge digital signatures or decrypt encrypted messages without knowledge of the appropriate private or decryption keys.
This framework assumes perfect cryptography, meaning that cryptographic primitives are treated as black boxes that behave ideally unless explicitly modeled otherwise (e.g., through key leakage rules). As such, Tamarin is highly effective for proving the logical soundness and cryptographic consistency of a protocol design. However, it does not account for physical-layer threats, implementation-level vulnerabilities, or side-channel attacks. Therefore, formal verification using Tamarin should be complemented with implementation testing and physical-layer analysis in practical deployments.
The complete Tamarin models for both protocols discussed in our work are included as part of our Github repository:

- `serverless.spthy` : models the serverless authentication protocol
- `authserver.spthy` : models the centralized server authentication protocol.

2 Protocol Modeling as Tamarin Rules

We begin by specifying the core primitives and function symbols that will be used throughout our Tamarin model. The `builtins` line imports several standard cryptographic operations, while the `functions` block defines custom symbolic functions required for our protocol logic.

```
begin
builtins: hashing, asymmetric-encryption, signing
functions: h/1, aenc/2,  adec/2, pk/1, sk/1, sign/2, verify/3, cert/3
```

Figure 1: Tamarin initialization

Below is a description of each defined function, modeled symbolically under the perfect cryptography assumption:

- `h(x)`: A hash function that returns a collision-resistant symbolic digest of input `x`.

- `aenc(x, y)`: Public-key encryption of message `x` using public key `y`.
- `adec(x, y)`: Private-key decryption of ciphertext `x` using private key `y`.
- `pk(x)`: Derives the public key corresponding to private key `x`.
- `sk(x)`: Derives the private key corresponding to public key `x`.
- `sign(x, y)`: Produces a digital signature of message `x` using private key `y`.
- `verify(sig, msg, pk)`: Verifies whether the signature `sig` corresponds to `msg` using public key `pk`.
- `cert(PK, PS, SIG)`: Models a digital certificate as a tuple containing the subject's public key `PK`, the issuer's public key `PS`, and the certificate's signature `SIG`. This abstraction reflects the essential structure of an X.509 certificate.

We also define a global equality restriction to enable symbolic term comparisons within our rules. This restriction enables symbolic rule guards to enforce logical conditions. For example, we can specify that a certificate is accepted only if the `verify()` function evaluates to `true`.

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

Figure 2: Equality Restriction

To enable pattern matching on certificates in our rules, we provide the following equation that allows Tamarin to destructure a certificate into its component fields.

```
equations:
  cert(PK, PS, SIG) = <PK, PS, SIG>
```

Figure 3: Certificate Equation Definition

We define first the rules that defines how various agents in the protocol—drones, certificate authorities (CAs), observers, and authentication servers—generate their public-private key pairs. Each rule uses the `Fr(~x)` construct to generate fresh private key material and then derives the corresponding public key via `pk(~x)`.

We mark the private key as secret using `Secret(~x)` and explicitly model that the public key is publicly known by emitting it with the `Out(pk(~x))` action. Moreover, keys belonging to trusted entities (e.g., the CA or Authentication Servers) are annotated with persistent facts such as `!TrustedAuthority`, `!CA`, or `!AuthServer`, enabling later rules to validate certificates and signatures originating from these roles.

```
rule generate_drone_keypair:
    [ Fr(~x) ]
  --[Secret(~x),Drone($A,pk(~x)), DroneKeyPair(~x,pk(~x))]->
  [ !Ltk($A, ~x), !Pk($A, pk(~x)), Out(pk(~x))]

  rule generate_CA_keypair:
    [ Fr(~x) ]
  --[Secret(~x), TrustedAuthority($S, pk(~x)), CAKeyPair(~x,pk(~x))]->
  [ !Ltk($S, ~x), !Pk($S, pk(~x)), Out(pk(~x)), !CA($S,pk(~x)) ]

rule generate_obesrver_keypair:
    [ Fr(~x) ]
  --[Secret(~x)]->
  [ !Ltk($A, ~x), !Pk($A, pk(~x)), Out(pk(~x)) ]

rule generate_AuthServer_keypair:
    [ Fr(~x) ]
  --[Secret(~x), TrustedAuthServer($S, pk(~x))]->
  [ !Ltk($S, ~x), !Pk($S, pk(~x)), Out(pk(~x)), !AuthServer($S,pk(~x)) ]
```

Figure 4: Key Pair Generation Rules

The protocol begins with a drone requesting a digital certificate from a trusted Certificate Authority (CA). This is necessary to establish a verifiable identity that can be authenticated by other parties in the system. In this step, the drone constructs and sends a Certificate Signing Request (CSR). This request contains the drone's public key and a signature that proves ownership of the corresponding private key. The request is encrypted with the CA's public key.

```
rule Drone_1:
  [ !CA($S,pkS),!Ltk($A,~x) ] -->
  [ Out(aenc(<pk(~x), sign(pk(~x),~x)>, pkS)), Sent_CSR($A, $S, pk(~x), pkS)]
```

Figure 5: Drone Certificate Signing Request Rule

After the drone has sent its encrypted Certificate Signing Request (CSR) to the network, the Certificate Authority (CA) must retrieve and process the message. This involves decrypting the request using the CA's private key and preparing the CSR for subsequent verification and certificate issuance.

```
rule CA_1:
  [!Ltk($S,~x), !CA($S, pk(~x)),
    In(aenc(<pkA, sign(pkA,sk(pkA))>, pk(~x)))]
    -->
  [Decrypt(adec(aenc(<pkA, sign(pkA,sk(pkA))>, pk(~x)), ~x))]
```

Figure 6: CA Receiving and Decrypting the CSR Rule

Once the Certificate Authority (CA) decrypts the Certificate Signing Request (CSR) sent by a drone, it must verify the authenticity of the request before issuing a certificate. Specifically, the CA checks

whether the CSR is signed using the private key corresponding to the public key included in the request. If the verification succeeds, the CA is assured that the request was genuinely issued by the owner of the public key, and proceeds to generate and sign a digital certificate and send it over the network.

```
rule CA_2:
  let SIG = snd(adec(aenc(<pkA, sign(pkA,sk(pkA))>, pk(~x)), ~x))
  PKA = fst(adec(aenc(<pkA, sign(pkA,sk(pkA))>, pk(~x)), ~x))
  in
  [!Ltk($S,~x),!CA($S, pk(~x)),Decrypt(adec(aenc(<pkA, sign(pkA,sk(pkA))>, pk(~x)), ~x))]
    --[Eq(verify(SIG,PKA,PKA),true),CreateCert(PKA, pk(~x),sign(PKA,~x))]->
  [Out(cert(PKA, pk(~x),sign(PKA,~x)))]
```

Figure 7: Certificate Generation by the CA Rule

After the CA has signed and sent the drone's certificate, the drone listens for the certificate on the network and validates its contents before accepting it. If the verification succeeds, the drone accepts the certificate and associates it with its identity by recording it as a persistent fact in the protocol.

```
rule Drone_2:
  [!Ltk($A,~x), !CA($S,pkS), In(cert(pk(~x), pkS,sign(pk(~x),sk(pkS)))), Sent_CSR($A, $S, pk(~x), pkS)]
    --[AcceptCert($A, cert(pk(~x), pkS,sign(pk(~x),sk(pkS))))]->
  [!Cert($A, cert(pk(~x), pkS,sign(pk(~x),sk(pkS))))]
```

Figure 8: Drone Certificate Acceptance and Storage Rule

2.1 Serverless RID Protocol Rules

Having defined the common rules shared by both protocols, we now proceed to the point where the model branches. We begin by presenting the serverless protocol.

After obtaining a valid certificate from the Certificate Authority (CA), the drone is able to immediately begin broadcasting its Remote ID (RID) message along with the corresponding certificate. Observers rely solely on the broadcasted information—specifically, the RID payload and its associated cryptographic signature—to verify the drone's identity and legitimacy without requiring any additional communication or infrastructure.

Due to payload size constraints (those imposed by the ASTM F3411 Remote ID message format) drones are often unable to transmit both the RID signature and the certificate within a single frame. To address this, the drone splits its broadcast into two distinct messages:

1. One message containing the certificate.
2. Another message containing the signed RID content.

```
rule Drone_3:
  [!Cert($A, cert(pk(~x), pkS,sign(pk(~x),sk(pkS)))),!Ltk($A,~x)] -->
  [Out(<$A, cert(pk(~x), pkS,sign(pk(~x),sk(pkS)))>)]
```

Figure 9: Drone Broadcasting its Certificate Rule

```
rule Drone_4:
  [!Cert($A, cert(pk(~x), pkS,sign(pk(~x),sk(pkS)))),!Ltk($A,~x)] --[SendRID($A, 'RID', sign('RID',~x))]->
  [Out(<$A, 'RID', sign('RID',~x)>)]
```

Figure 10: Drone Broadcasting its signed RID Rule

Observers on the network passively listen to these broadcasts. Upon reception, the observer verifies and stores them for subsequent validation.

```
rule Observer_1:
  [!CA($S,pkS), In(<$A, cert(pkA, pkS,sign(pkA,sk(pkS)))>)] --[AcceptCert($A, cert(pkA, pkS,sign(pkA,sk(pkS))))]->
  [GetCert($A, cert(pkA, pkS,sign(pkA,sk(pkS))))]
```

Figure 11: Observer Receiving and Storing a Drone Certificate Rule

```
rule Observer_2:
  [In(<$A, 'RID', SIG>) ] -->
  [GetRID($A, 'RID',SIG)]
```

Figure 12: Observer Receiving a Signed RID Message Rule

When an observer has collected both the certificate and the signed RID message from a given drone, it proceeds to verify the authenticity of the RID content.

```
rule Observer_3:
  let CERT = cert(pkA, pkS,sign(pkA,sk(pkS)))
  PKA = fst(fst(CERT))
  in
  [GetCert($A, cert(pkA, pkS,sign(pkA,sk(pkS)))), GetRID($A, 'RID', SIG) ]
  --[Eq(verify(SIG,'RID',PKA),true), AcceptCert($A,CERT)]->[VerifedRID($A, 'RID',CERT,SIG)]
```

Figure 13: Observer Verifying a Signed RID Message Rule

And the observer accepts only those RID messages that have been cryptographically verified.

```
rule Observer_4:
  [VerifedRID($A, 'RID',CERT,SIG)]--[RIDAccepted($A, 'RID',CERT,SIG)]->[]
```

Figure 14: Observer Accepting a Verified RID Message Rule

2.2 Centralized Authentication Server Protocol Rules

In the centralized authentication protocol, each drone must first register its certificate with a trusted authentication server. This allows the server to later validate RID messages signed by the drone by associating its identity with a known certificate. The registration is performed by securely transmitting the certificate to the Authentication Server.

```
rule Drone_3:
  [!Cert($A, cert(pk(~x), pkS,sign(pk(~x),sk(pkS)))),!Ltk($A,~x), !AuthServer($T, pkT)] -->
  [Out(aenc(<$A, cert(pk(~x), pkS,sign(pk(~x),sk(pkS)))>,pkT)), !RegisterDrone($A, cert(pk(~x), pkS,sign(pk(~x),sk(pkS))))]
```

Figure 15: Drone Registering Its Certificate with the Authentication Server Rule

Once the drone sends its encrypted certificate to the authentication server, the Authentication Server must decrypt, validate, and register the certificate—but only if it was signed by a trusted CA. This ensures that only drones with legitimate identities are admitted into the system.

The verification process is modeled across two rules: the first for decrypting the message, and the second for validating and storing the certificate. A persistent fact `!DroneInAuthServer(ID, CERT)` is declared to indicate that the drone, along with its corresponding certificate, has been correctly registered and recognized by the authentication server.

```
rule AuthServer_1:
    [!Ltk($T,~x), !CA($S,pkS), !AuthServer($T,pk(~x)), In(aenc(<$A, cert(pkA, pkS,sign(pkA,sk(pkS)))>,pk(~x)))] -->
    [Decrypt(aenc(<$A, cert(pkA, pkS,sign(pkA,sk(pkS)))>,pk(~x)))]

rule AuthServer_2:
  let CERT = snd(adec(aenc(<$A, cert(pkA, pkS,sign(pkA,sk(pkS)))>,pk(~x)), ~x))
  ID = fst(adec(aenc(<$A, cert(pkA, pkS,sign(pkA,sk(pkS)))>,pk(~x)), ~x))
  in
  [!Ltk($T,~x), !AuthServer($T,pk(~x)),Decrypt(aenc(<$A, cert(pkA, pkS,sign(pkA,sk(pkS)))>,pk(~x))),!CA($S,pkS)]
  --[RegCert(ID,CERT), AcceptCert($A, cert(pkA, pkS,sign(pkA,sk(pkS))))]->
  [!DroneInAuthServer(ID,CERT)]
```

Figure 16: Authentication Server Processing Drone Certificate Registration Rules

Now that the drone has completed its registration process, it is authorized to operate and can begin broadcasting its Remote ID (RID) message. This broadcast is passively received by any observer in range.

```
rule Drone_4:
  [!RegisterDrone($A, cert(pk(~x), pkS,sign(pk(~x),sk(pkS))),!Ltk($A,~x)] --[SendRID($A, 'RID', sign('RID',~x))]->
  [Out(<$A, 'RID', sign('RID',~x)>)]
```

Figure 17: Drone Broadcasting a Signed RID Message After Registering to Authentication Server Rule

On the receiving end, an observer listens for any such broadcast and records the RID message and its corresponding signature. However, unlike the serverless protocol, where verification occurs locally, the observer in the centralized server-based protocol must verify this RID by checking the central registry maintained by the authentication server.

```
rule Observer_5:
   [In(<$A, 'RID', SIG>) ] -->
   [GetRID($A, 'RID',SIG)]
```

Figure 18: Observer Receiving a Signed RID Message Rule

To interact with the authentication server's services, an observer must first complete a registration process. The observer begins by generating a fresh password `~pwd`. This password is then hashed using a one-way hash function `h(~pwd)` to ensure that even if the message is intercepted, the actual password remains protected. The observer then creates a structured payload consisting of the tag 'Register', its identity `$A`, and the hashed password. This payload is encrypted using the public key of the authentication server and emitted onto the network.

```
rule Observer_1:
  [Fr(~pwd), !Ltk($A, ~x), !AuthServer($T,pkT)] -->
  [Out(aenc(<<'Register'>,<$A, h(~pwd)>>,pkT)),!UserInAuthService($A,~pwd,$T)]
```

Figure 19: Observer Registration with the Authentication Server Rule

After the observer's registration message is sent, the authentication server processes the request through two sequential rules. First, the server detects the arrival of the encrypted message and decrypts it using its long-term private key. Once decrypted, the server extracts and parses the registration payload, saving the observer's identity and password to `ID` and `PWD` respectively. It then commits the persistent

fact `!ObserverInAuthServer(ID, PWD)` to indicate that the observer has been successfully registered and is now recognized by the system.

```
rule AuthServer_3:
  [!Ltk($T, ~x), !AuthServer($T,pk(~x)),In(aenc(<<'Register'>,<$A, pwd>>,pk(~x)))] -->
  [Decrypt(aenc(<<'Register'>,<$A, pwd>>,pk(~x)))]

rule AuthServer_4:
  let CRED = snd(adec(aenc(<<'Register'>,<$A, pwd>>,pk(~x)),~x))
  ID = fst(CRED)
  PWD = snd(CRED)
  in
  [!Ltk($T, ~x),!AuthServer($T,pk(~x)),Decrypt(aenc(<<'Register'>,<$A, pwd>>,pk(~x)))] -->
  [!ObserverInAuthServer(ID,PWD)]
```

Figure 20: Authentication Server Processing Observer Registration Rules

Now, whenever the observer wishes to access services from the authentication server, it must first initiate a login session by sending a login request. This request includes the observer's identity, the hashed version of its password, and the public key of the device it is currently using. Including the device's public key serves to bind the login session specifically to that device. The message is then encrypted with the authentication server's public key.

```
rule Observer_3:
  [!UserInAuthService($A,~pwd,$T), !Ltk($A, ~x), !AuthServer($T,pkT)] -->
  [Out(aenc(<<'Login',$A>,<h(~pwd),pk(~x)>>,pkT))]
```

Figure 21: Observer Initiating a Login Session Rule

Upon receiving the login request, the authentication server first verifies whether the observer's identifier and hashed password correspond to a valid registered observer in its records. If the credentials match, the server proceeds to generate a login token by signing the observer's credentials using its own private key. This token serves as proof that the observer has been authenticated. To ensure that only the requesting device can use this token, the server encrypts the token using the public key of the observer's current device—extracted from the login request—and transmits it back over the network.

```
rule AuthServer_5:
  [!Ltk($T, ~x),!AuthServer($T,pk(~x)), In(aenc(<<'Login',$A>, <PWD,pkA>>,pk(~x)))] -->
  [Decrypt(aenc(<<'Login',$A>, <PWD,pkA>>,pk(~x)))]

rule AuthServer_6:
  let DEC = adec(aenc(<<'Login',ID>, <PWD,pkA>>,pk(~x)),~x)
  CRED = <ID, PWD>
  TOKEN = sign(CRED, ~x)
  PKA = snd(snd(DEC))
  in
  [!Ltk($T, ~x),!AuthServer($T,pk(~x)),Decrypt(aenc(<<'Login',$A>, <PWD,pkA>>,pk(~x))),!ObserverInAuthServer(ID,PWD)]
  --[IssueToken(TOKEN, $T)]->
  [Out(aenc(<'TOKEN',TOKEN>,PKA))]
```

Figure 22: Authentication Server Handling Observer Login and Token Issuance Rule

Once the observer receives the encrypted login token from the authentication server, it decrypts the message using its private key to retrieve the token. The observer then validates the token by verifying the server's signature over its own identity and password hash using the public key of the authentication server. If this verification succeeds, the token is considered valid and is stored locally by the observer for use in subsequent authenticated interactions with the server.

```
rule Observer_4:
  [!Ltk($A, ~x), In(aenc(<'TOKEN',TOKEN>,pk(~x))),!UserInAuthService($A,~pwd,$T),!AuthServer($T,pkT)]
  --[Eq(verify(TOKEN,<$A, h(~pwd)>,pkT),true)]->
  [Token($T, TOKEN)]
```

Figure 23: Observer Verifying and Storing a Received Login Token Rule

The observer is logged in to the authentication server and possesses a valid token; finally, it can now verify any received RID message. To do so, the observer sends the RID message along with the login token—encrypted using the server's public key—to the authentication server. After sending the request, the observer waits for a verification response from the server.

```
rule Observer_6:
  [GetRID($D, 'RID',SIG), !Ltk($A, ~x), Token($T, TOKEN), !AuthServer($T,pkT), !UserInAuthService($A,~pwd,$T)]
  --[UseToken(TOKEN)]->
  [Out(aenc(<<$D, 'RID', SIG>, <TOKEN, pk(~x)>>, pkT)), WaitForRIDResponse($D, 'RID', SIG)]
```

Figure 24: Observer Sending RID Verification Request to the Authentication Server Rule

When the authentication server receives a RID verification request, it first decrypts the message and verifies the observer's login token by checking its signature against the stored credentials. If the token is valid and the observer is recognized, the server then checks whether the drone identifier contained in the RID corresponds to a previously registered drone. It retrieves the certificate associated with that drone and uses it to verify the digital signature of the RID message.

If the RID signature is valid and the certificate matches the expected identity, the server concludes that the RID is authentic. It then responds to the observer with a verification message containing the status together with a cryptographic proof of validity (a server-generated signature of its answer). This proof allows the observer to confirm that the authentication server attests to the legitimacy of the RID message.

```
rule AuthServer_7:
  [!Ltk($T, ~x), !AuthServer($T,pk(~x)), In(aenc(<<$D, 'RID', SIG>, <sign(<ID, PWD>,~x),pkA>>, pk(~x))),!ObserverInAuthServer(ID,PWD)]
  -->
  [Decrypt(aenc(<<$D, 'RID', SIG>, <sign(<ID, PWD>,~x), pkA>>, pk(~x)))]

rule AuthServer_8:
  let DEC = adec(aenc(<<$D, 'RID', SIG>, <sign(<ID, PWD>,~x),pkA>>, pk(~x)), ~x)
  RID = fst(DEC)
  PKA = snd(snd(DEC))
  in
  [!Ltk($T, ~x),!AuthServer($T,pk(~x)), Decrypt(aenc(<<$D, 'RID', SIG>, <sign(<ID, PWD>,~x), pkA>>, pk(~x))), !ObserverInAuthServer(ID,PWD),
   !DroneInAuthServer($D, CERT)]
  --[Eq(verify(SIG,'RID',fst(CERT)),true)]->
  [Out(<<$D, 'RID', SIG>, <'VALID',sign('VALID',~x),CERT>>)]
```

Figure 25: RID Verification and Response by the Authentication Server Rules

After the observer receives the server's verification response—stating that the RID signature is valid and has been cryptographically authenticated using the registered certificate—it decrypts the message and marks the RID as accepted. This concludes the authentication flow: the observer now considers the received RID message trustworthy and originating from a verified drone.

```
rule Observer_7:
  [!Ltk($A, ~x), !AuthServer($T,pkT),In(<<$D, 'RID', SIG>, <'VALID',PROOF,CERT>>), WaitForRIDResponse($D, 'RID', SIG)]
  --[Eq(verify(PROOF,'VALID',pkT),true), RIDAccepted($D,'RID',CERT,SIG)]-> []
```

Figure 26: Observer Confirmation of RID Verification Result Rule

3 Tamarin Formal Lemmata Specifications

In Tamarin Prover, protocol security goals are expressed as Lemmata. Each lemma formalizes a desired property, and Tamarin attempts to prove whether it holds across all possible execution traces (all sequences of events and message exchanges under adversarial control). The lemmata serve as the backbone

of our formal analysis, each corresponding to a core security property we want to prove our protocol possesses: liveness, secrecy, authenticity, and integrity.

Lemma 1 (Protocol Liveness): It ensures protocol liveness, that the system can actually reach a valid end state through a complete execution. This kind of lemma does not directly represent a security goal, but it is critical to establishing the credibility of the other lemmata; After all, it is of no use proving that our protocol is secure if it is impossible to run. In our case, it confirms that an observer can eventually verify and accept a signed RID message as authentic.

```
lemma Protocol_Liveness:
  exists-trace
    "Ex A CERT SIGRID #i. RIDAccepted(A,'RID',CERT,SIGRID) @ i"
```

Figure 27: Protocol Liveness Lemma

Lemma 2 (Key Secrecy): It guarantees that the drone's private key remains secret throughout all protocol executions. We model this by asserting that if a value is declared `Secret`, then it must never be derivable by the adversary. This is a foundational requirement as accidental leakage of the signing key would allow adversaries to forge RID messages and impersonate the drone.

```
lemma Secrecy:
  "All x #i.
    Secret(x) @i ==> not (Ex #j. K(x)@j)"
```

Figure 28: Private Key Secrecy Lemma

Lemma 3 (Certificate Integrity): It ensures that only a trusted Certificate Authority (CA) can issue a certificate. Whenever a certificate is accepted as legitimate, it must have been signed by a public key associated with a designated trusted authority.

```
lemma Cert_From_CA_Only:
  "All A PK PS SIG #i. AcceptCert(A,<PK, PS,SIG>)  @ i
     ==>  (Ex S #j. TrustedAuthority(S, PS) @ j)"
```

Figure 29: Certificate Integrity Lemma

Lemma 4 (JWT Authenticity): It only applies to the centralized protocol; the JWT Authenticity lemma asserts that every JWT token used in RID verification must have been issued by the authentication service to a registered observer. This prevents the adversary from crafting or reusing tokens to impersonate legitimate observers. Formally, this means that for any trace in which a JWT token is used, there must exist a preceding event in which a trusted authentication server issued and signed the given token.

```
lemma JWT_Authenticity:
  "All TOKEN #i.
     UseToken(TOKEN) @ i
     ==>
     (Ex #j1 #j2 T sK CRED.
        TOKEN = sign(CRED, sK)
        & IssueToken(TOKEN, T) @ j1
        & TrustedAuthServer(T,pk(sK)) @j2
     )"
```

Figure 30: JWT Authenticity Lemma

Lemma 5 (Remote ID Integrity): this ensures that any broadcasted RID message accepted by the observer was signed using the private key corresponding to the drone's certified public key, and that the message was not altered by the adversary.

```
lemma RID_Integrity:
  "All A PK PS SIG SIGRID #i.
     RIDAccepted(A, 'RID', <PK, PS, SIG>, SIGRID) @ i
     ==>
    (Ex x meta.
        SIGRID = sign('RID', x) &
        PK = <pk(x),meta>)"
```

Figure 31: RID Integrity Lemma

Lemma 6 (Injective Agreement): It establishes injective agreement, which ensures that both communicating entities agree on a unique instance of session parameters and identities. In the context of one-way communication (broadcast RID messages from a drone to an observer), this property implies that the observer accepts an RID message only if it can verify both the integrity of the message and the authenticity of its origin, ensuring that it was indeed sent by the drone it claims to represent. We formally state it as if two RID acceptances happen with the same drone identity, certificate, and signature, then they must correspond to the same protocol instance (i.e., the same event index, `#i1 = #i2`).This ensures that each accepted RID is uniquely tied to one unique drone, ruling out replayed messages.

```
lemma Injective_Agreement:
  "All D CERT SIG #i1 #i2.
     RIDAccepted(D,'RID',CERT,SIG) @ i1
    & RIDAccepted(D,'RID',CERT,SIG) @ i2
    ==> (#i1 = #i2)"
```

Figure 32: Injective Agreement Lemma

4 Tamarin Verification Trace Results

4.1 Serverless Protocol Trace Results

To confirm that our protocol can successfully execute its core functionality, we verified a liveness property using the Tamarin Prover. Specifically, we proved the lemma `Protocol_Liveness`, which asserts that there exists a valid execution trace in which a Remote ID message, signed and certified by a legitimate drone, is ultimately accepted by an observer. Figure 33 illustrates the verified trace that satisfies this lemma. The execution begins with the generation of a certificate authority (CA) key pair, followed by the drone constructing a certificate containing its public key signed by the CA. The drone then broadcasts its signed RID message along with the certificate. The observer receives both artifacts, verifies the certificate chain, and checks the signature over the RID message. Upon successful verification, the observer issues an `RIDAccepted` event, confirming that the drone's broadcast was accepted. This trace confirms that the protocol is live — it supports a feasible end-to-end interaction from credential generation through to observer acceptance — and thus is not over-constrained or trivially flawed. While this result does not imply resistance to attacks, it establishes that the protocol can operate successfully in honest executions.

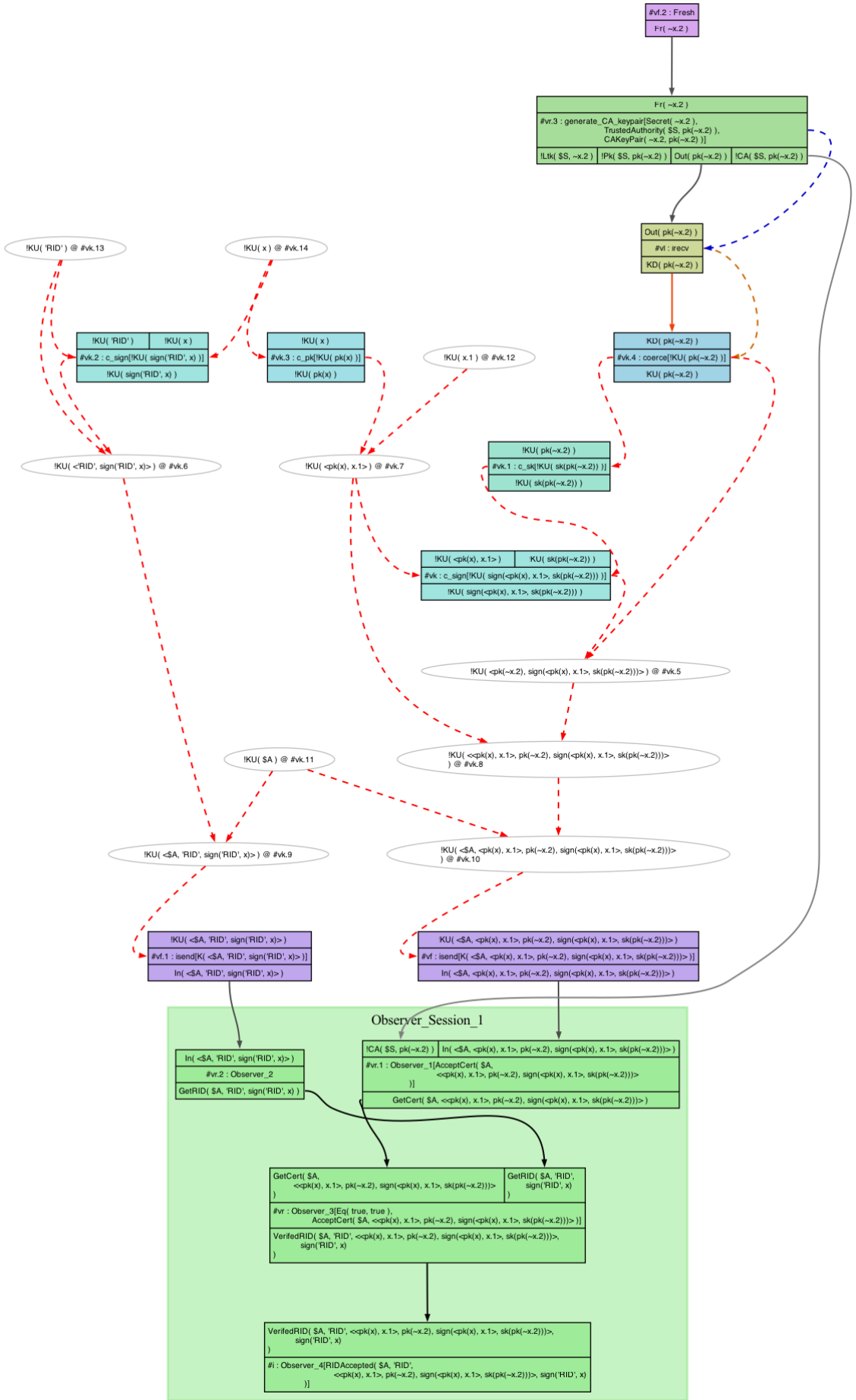
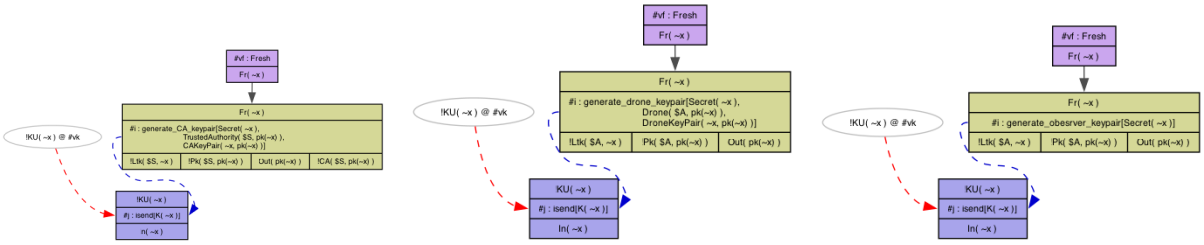


Figure 33: Tamarin Prover Execution Trace for the **Protocol Liveness** Lemma

To validate the secrecy of private keys in our protocol, we proved the lemma Secrecy, which states that if a value `x` is declared secret (i.e., `Secret(x)` occurs), then no adversary can ever learn `x` (i.e., there exists no time `j` such that `K(x) @ j`). This is a core confidentiality property that guarantees the long-term secrecy of private key material. The Tamarin Prover successfully verified this lemma for all key generation processes in the protocol.

Figure 34 illustrates three representative execution traces, each corresponding to the generation of a key pair by a different role. In each case, a fresh secret `~x` is created, and the corresponding public key `pk(~x)` is shared via output `Out(...)`, while the secret remains internally stored `!Ltk(...)`. The protocol ensures that this private key is never leaked or exposed to the attacker, which is confirmed by the absence of any `K(~x)` fact in the trace. The red dashed line from `!KU(~x)` (intruder knowledge update) halts at a dead-end, visually confirming that the intruder never obtains the private value. This verified secrecy property ensures that the core trust assumptions of the protocol (e.g., digital signature security and identity authenticity) hold against symbolic adversaries under the Dolev-Yao model.



(a) CA keypair generation (b) Drone keypair generation (c) Observer keypair generation

Figure 34: Tamarin Prover trace fragments for the proven **Secrecy** lemma. Each subfigure demonstrates that although the private key component `~x` is used to derive a public key, the intruder does not gain access to `~x`. The red dashed arrows denote blocked knowledge acquisition attempts by the adversary, confirming that the secret remains undisclosed.

To formally ensure that all accepted certificates in our protocol are issued by legitimate certificate authorities (CAs), we verified the lemma `Cert_From_CA_Only`. This lemma asserts that if any agent `A` accepts a certificate tuple `<PK, PS, SIG>`, then there must exist some `S` such that `PS` is the public key of a trusted authority `S`. This condition is fundamental to establishing trust in the public-key infrastructure used by our protocol.

Figures 35, 36, and 37 illustrate representative counterexample search traces where Tamarin attempted to falsify this lemma but failed — solving each case by contradiction. In each trace, the certificate acceptance event `AcceptCert(...)` is clearly recorded; however, due to the absence of a valid `TrustedAuthority` fact binding the public key `PS` used in the certificate signature, the system arrives at a contradiction. The intruder's ability to construct forged tuples involving `sign(pk, sk(pk))` is shown via red dashed inference paths, but these constructions do not satisfy the requirement of having a legitimate CA in the trace. This proves that under our model assumptions, forged certificates — even if structurally valid — will not be accepted unless their signing key originates from a CA.

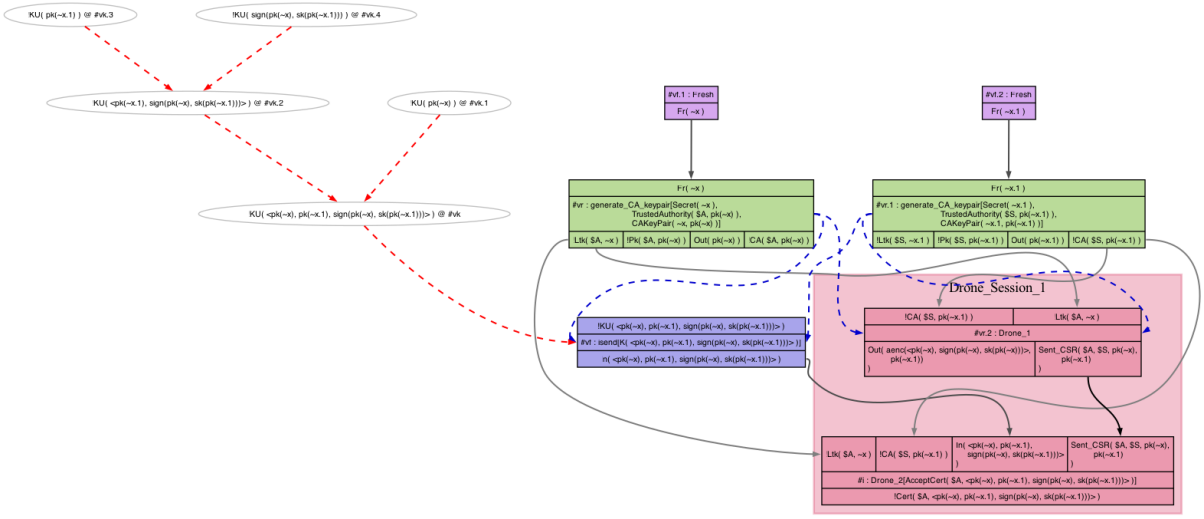


Figure 35: Tamarin trace for a drone session attempting to accept a certificate. The proof is resolved by contradiction, as the certificate signer `PS` used in the signature does not correspond to any declared Trusted Authority.

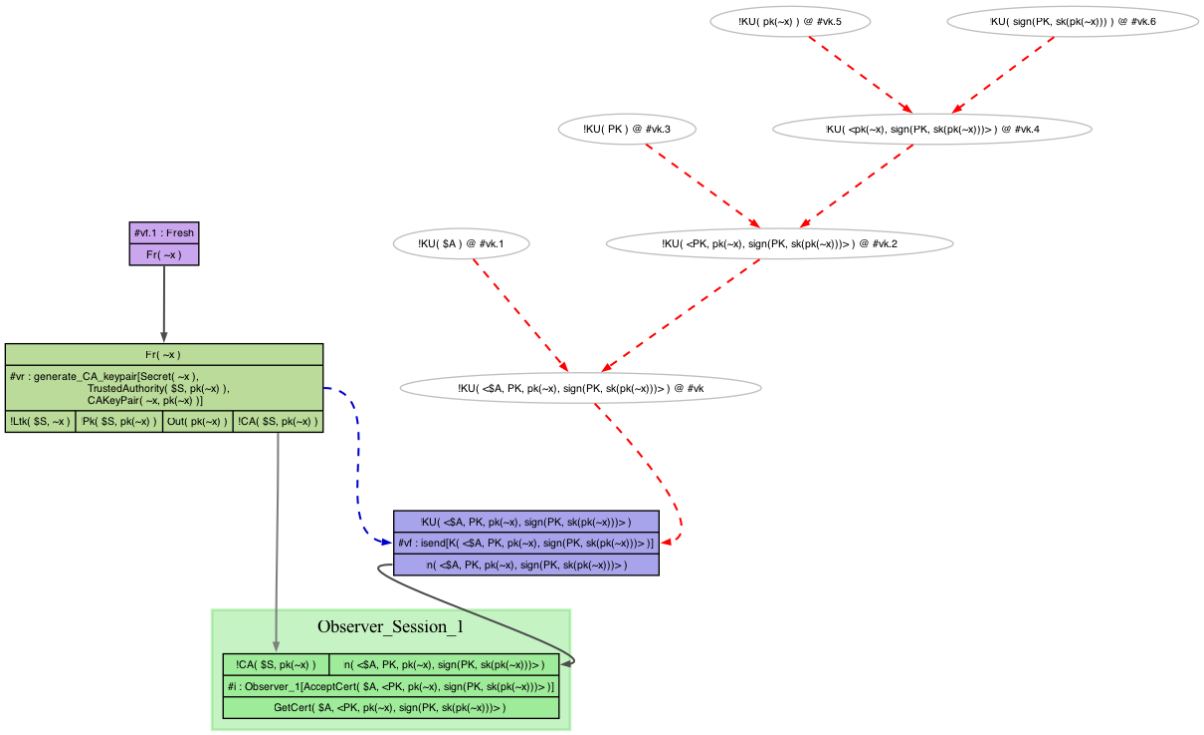


Figure 36: Observer session trace showing certificate acceptance with no associated `TrustedAuthority` fact for the signer key. The adversary's construction is invalidated by lack of CA authentication.

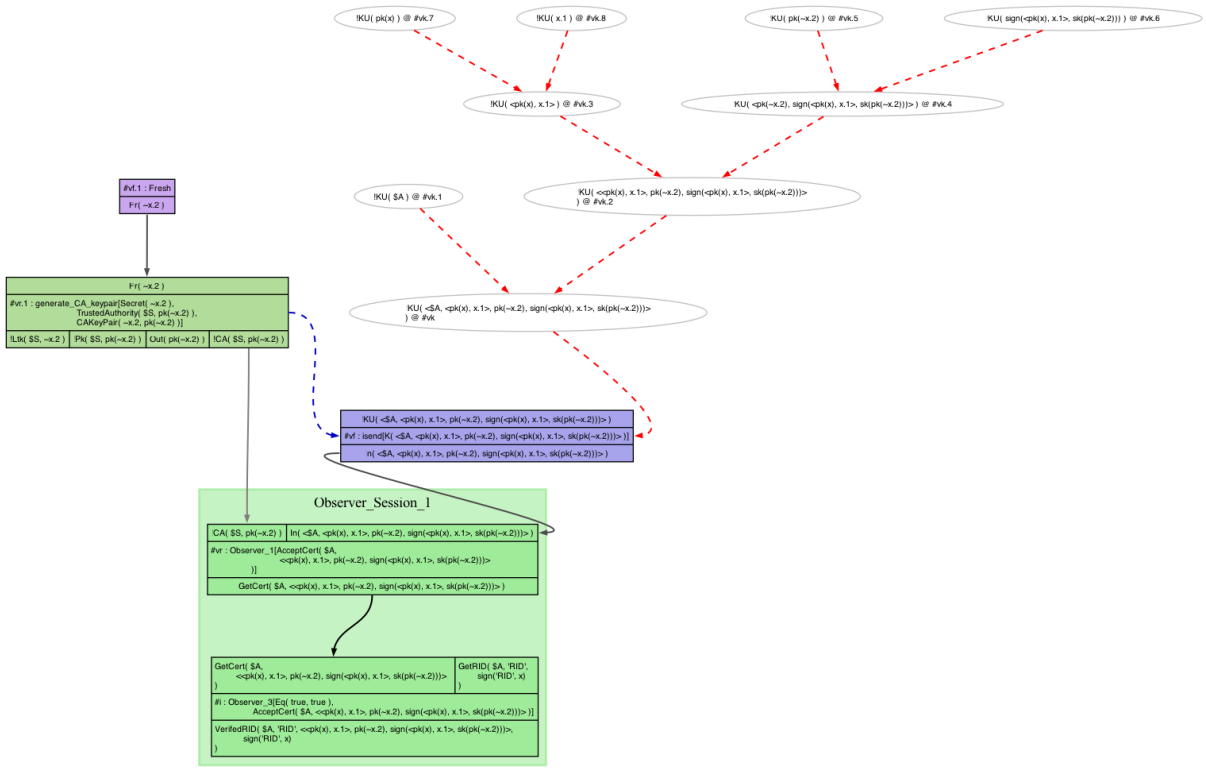


Figure 37: A more complex observer verification path where the certificate's signature is derived by the intruder, but again rejected due to absence of a trusted CA for the signing key.

To verify that Remote ID (RID) messages cannot be spoofed or mismatched with forged keys, we proved the lemma `RID_Integrity`. This lemma asserts that if an observer accepts a signed RID message and its associated certificate `<PK, PS, SIG>`, then the signature `SIGRID` must have been produced using the private key corresponding to the public key `PK`. Specifically, there must exist a value `x` (i.e., the secret signing key) such that `SIGRID = sign('RID', x)` and `PK = <pk(x), meta>`, where meta may contain additional data such as key metadata or encoding.

The trace in Figure 38 demonstrates Tamarin's failed attempt to contradict this property. The adversary accumulates all relevant inputs, including the certificate and RID message, attempting to forge a valid `SIGRID`. However, the observer's internal rule for verifying RID `VerifiedRID(...)` tightly links the signature to a legitimate key pair derived from the original certificate. Because the private key `x` used for signing is only generated internally and never leaked to the intruder, Tamarin cannot derive an alternate value of `SIGRID` that would still be accepted by the observer. This leads to a contradiction when trying to falsify the lemma, thereby confirming its validity.

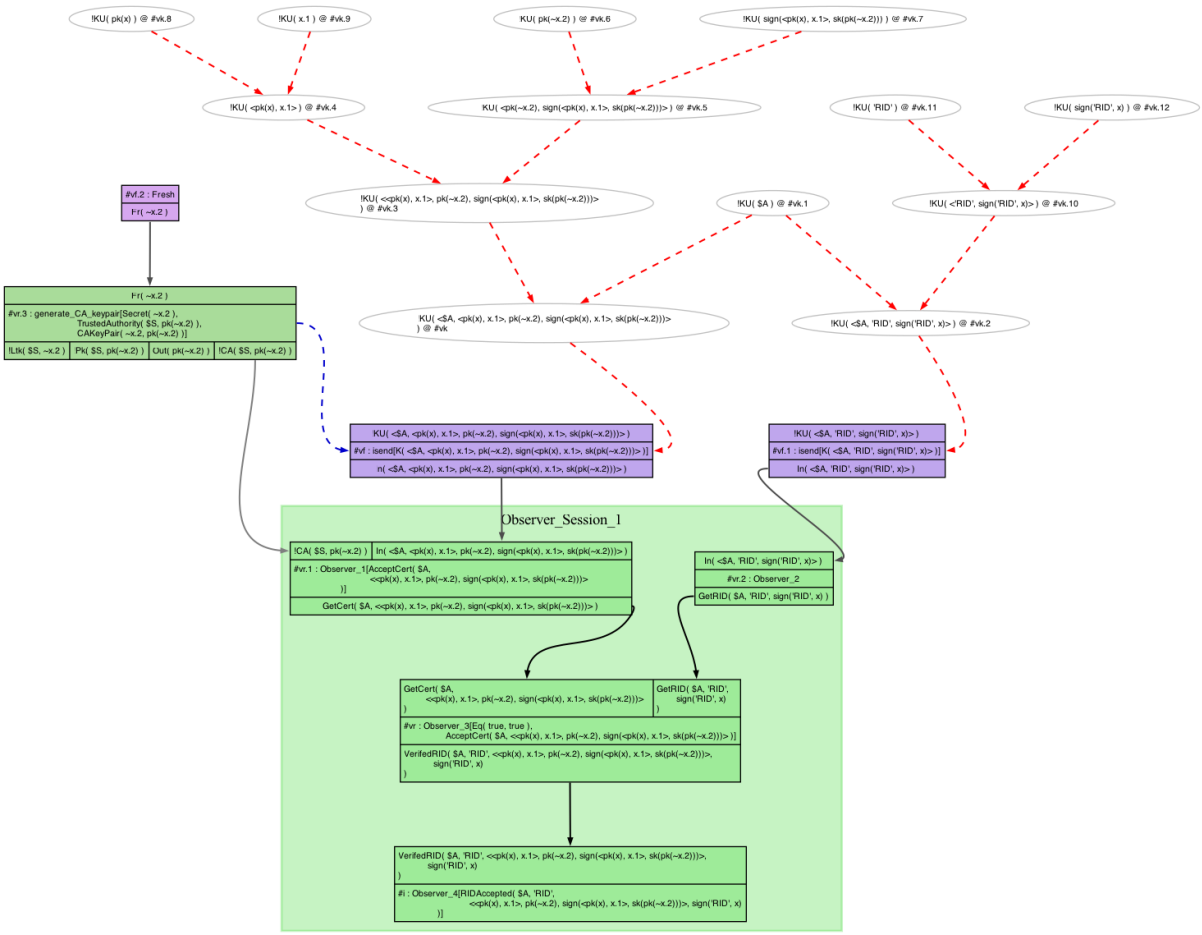


Figure 38: Proof trace for the **RID_Integrity** lemma. The observer session (green box) successfully verifies a signed RID message based on a certificate issued by a trusted CA. The intruder's attempts to forge `SIGRID` (shown with red dashed inference arrows) are unsuccessful, and no alternate signing value can produce a valid match. Tamarin resolves the attempted contradiction, thereby confirming that `SIGRID` must correspond to a key contained within the accepted certificate.

The `Injective_Agreement` lemma aims to enforce a strong authenticity guarantee: whenever an observer accepts a signed Remote ID (RID) message, that message must have been generated by a drone `A` with key `PK`, and the drone must have actually sent that exact `SIGRID` value. This prevents message replays or identity spoofing — requiring a cryptographic and behavioral match between sender and receiver perspectives.

However, in the context of the serverless Remote ID protocol, this property fails. As shown in Figure 39, the Tamarin Prover generates a counterexample trace in which the same signature and certificate are accepted from two different sources: at distinct trace points `#i1` $\neq$ `#i2`. This directly violates injective agreement, rendering the lemma false.

The core reason for this failure lies in the nature of broadcast communication. Once a signed RID message is broadcast and captured, an adversary can replay it freely.

In the trace, the red dashed arrows clearly show how the intruder reuses the same `<D, 'RID', SIG>` tuple, obtained from one broadcast, to trigger acceptance in multiple observer sessions. This reveals a fundamental limitation of broadcasting schemes: while message authenticity (via signature verification) is preserved, replayability undermines injective agreement.

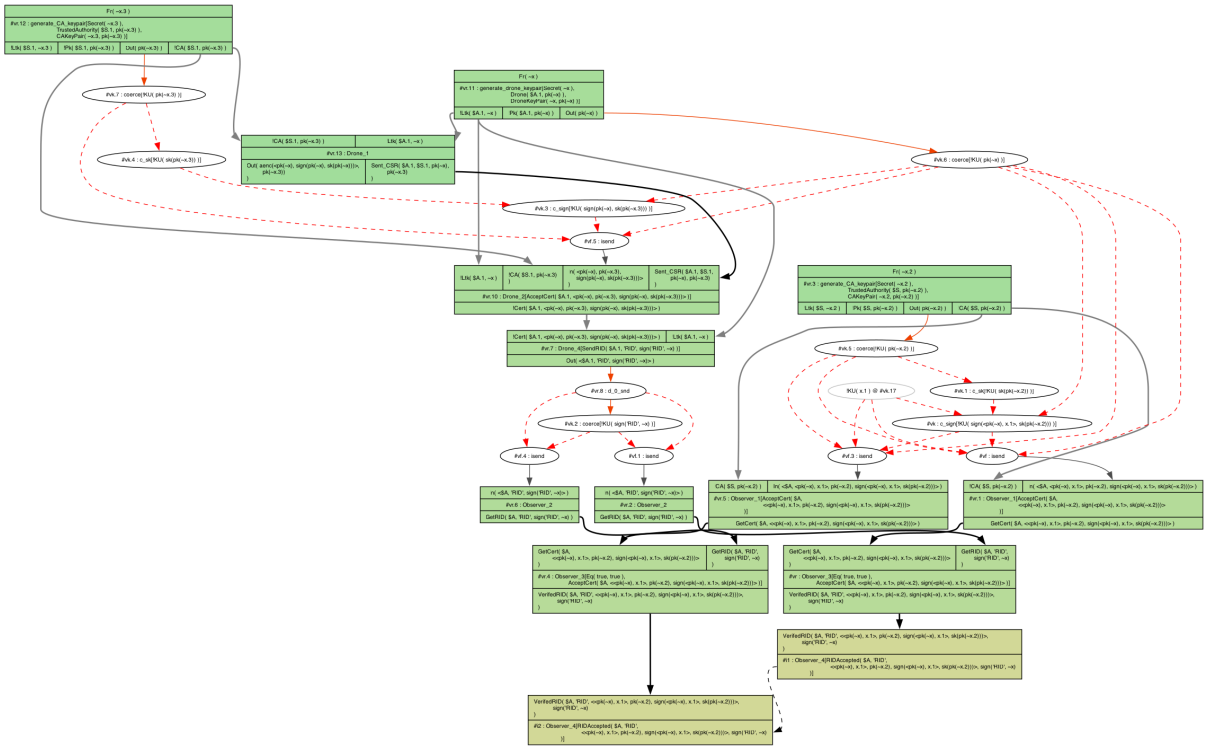


Figure 39: Tamarin Prover trace disproving the **Injective_Agreement** lemma. The same signed RID message is accepted from two distinct sources by observers, violating the injectivity condition.

4.2 Centralized Authentication Server Protocol Trace Results

The `Protocol_Liveness` lemma asserts that there exists at least one valid execution trace in which an observer successfully accepts a signed RID message from a drone. The acceptance of a signed RID `RIDAccepted(...)` means that a drone has successfully registered and transmitted an RID message, which an observer has verified using cryptographic credentials issued by a trusted Certificate Authority (CA) and authenticated by an Authorization Server.

Figure 40 illustrates the verified execution trace that satisfies the liveness lemma in the Authentication Server-based Remote ID protocol. The trace involves multiple entities: a drone generates its key pair and registers its credentials with the authentication server, the observer logs in and obtains a valid token, and eventually receives and verifies the RID message broadcast by the drone. The observer performs full validation of the certificate and signature using the authentication server's verification mechanisms. The final acceptance `RIDAccepted` occurs only after all preconditions — including token use, certificate validity, and RID signature verification — are satisfied.

The successful trace confirms that the protocol permits a complete end-to-end flow: drone registration, certificate issuance, observer login and token issuance, RID broadcast, and RID acceptance.

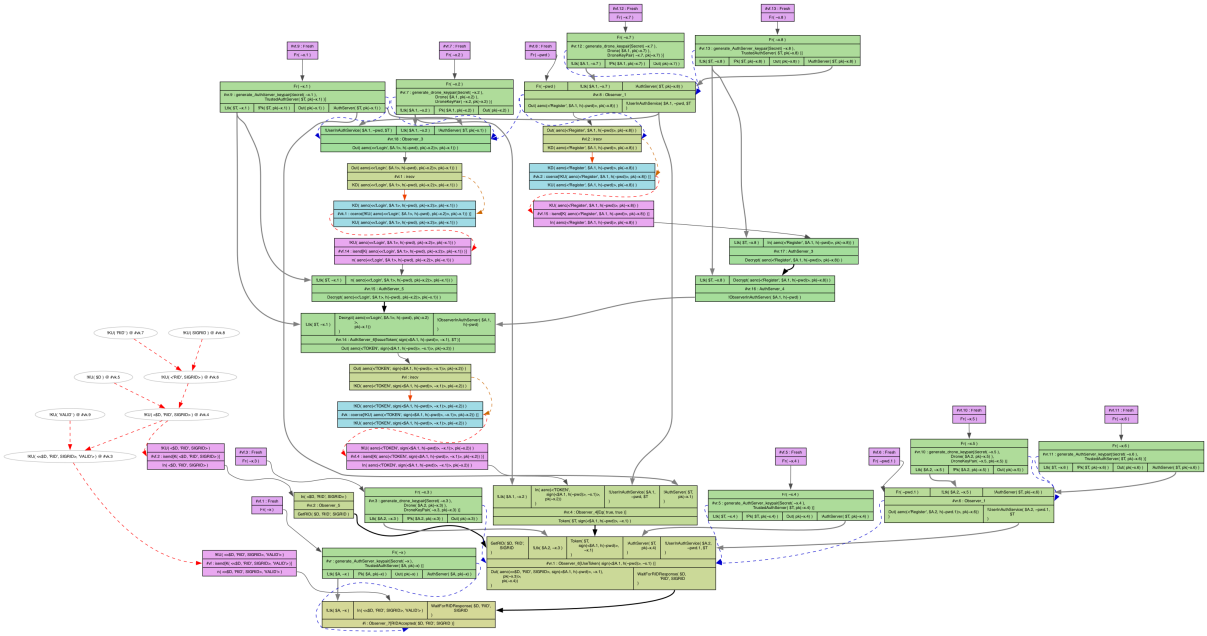
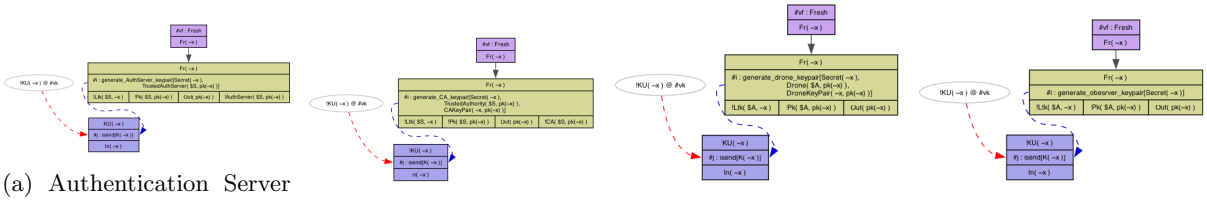


Figure 40: Tamarin Prover execution trace verifying the **Protocol_Liveness** lemma in the Authentication Server-based Remote ID protocol

To ensure the confidentiality of long-term keys across all critical roles, we verified the `Secrecy` lemma under the centralized authentication Server-based RID authentication model. The lemma guarantees that if any key `x` is marked `Secret(x)`, then the intruder will never learn its value — formally, there does not exist any point in the trace where `K(x)` holds.

Figure 41 presents four subfigures showing the Tamarin traces for each entity's key generation rule. The trace confirms that in every case — whether for the server, CA, drone, or observer — the secret key is freshly generated, internally stored via `!Ltk(...)`, and only the corresponding public key `pk(x)` is revealed externally. Despite the adversary's attempt to infer `x` (shown via red dashed lines terminating at `!KU(x)`), no derivation is possible. These traces collectively confirm that all entities maintain perfect secrecy of their private keys under the Dolev-Yao model.



(a) Authentication Server keypair (b) CA keypair (c) Drone keypair (d) Observer keypair

Figure 41: Tamarin Prover trace fragments showing the verification of the **Secrecy** lemma in the Authentication Server protocol. Each entity generates a fresh secret key ∼x, stores it persistently, and only shares the public component. The adversary's derivation attempts (red dashed arrows) terminate without success, confirming that no secret key is leaked.

To make sure of the authenticity of issued certificates, we proved the lemma `Cert_From_CA_Only`, which enforces that no participant should accept a certificate `<PK, PS, SIG>` unless the signing public key `PS` is associated with a legitimate certificate authority (CA), modeled as `TrustedAuthority(S, PS)`. This prevents adversaries from injecting self-signed or rogue certificates into the system.

Figures 42 and 43 present two trace results for different protocol roles. In the first trace (Authentication Server session), Tamarin attempted to falsify the lemma but reached a contradiction: the certificate is accepted, yet no valid CA key `PS` was ever declared as trusted, thus violating the requirement. This invalid trace conclusively proves the lemma by contradiction; No trace can exist that violates the lemma without leading to inconsistency.

The second trace (Drone session) illustrates a valid flow where a drone accepts a certificate that was correctly signed by a key PS known to be associated with a trusted CA. The certificate `pk( x), PS,`

`sign(pk(~x), sk(PS))>` is constructed and verified, with the corresponding `TrustedAuthority(S, PS)` fact present in the trace. This supports the lemma's condition in a live protocol run.

Together, these traces confirm that only certificates signed by CAs are ever accepted.

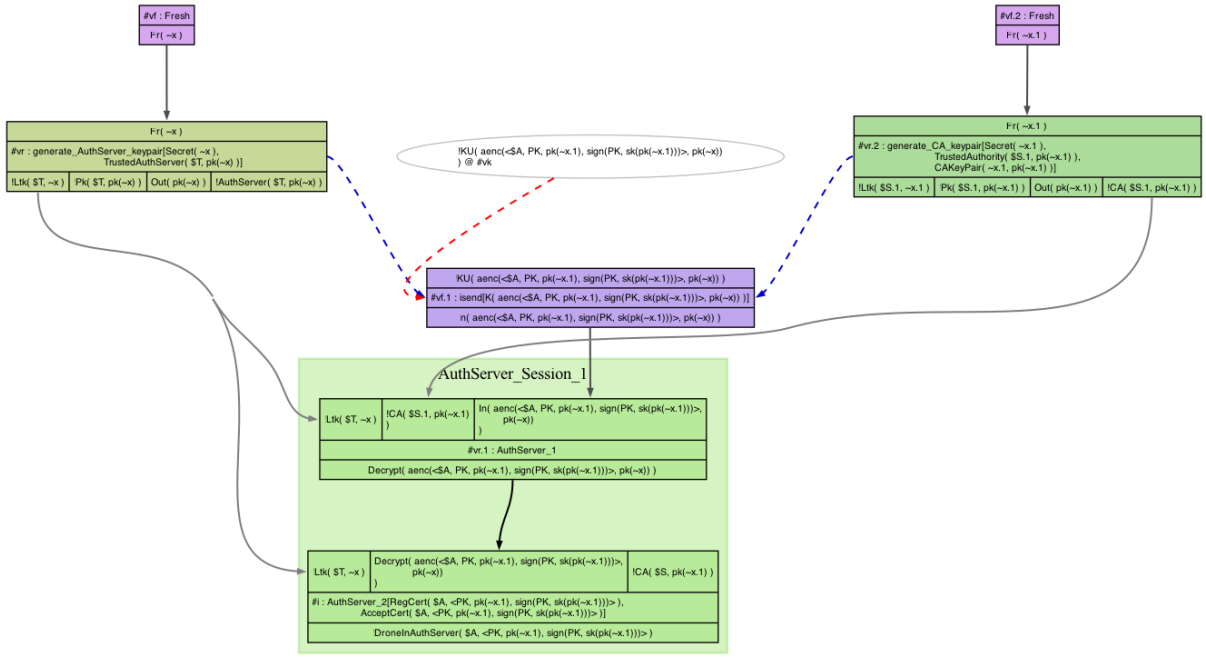


Figure 42: Tamarin trace disproving the existence of a rogue certificate path in the AuthServer case. The `AcceptCert` action occurs without a `TrustedAuthority` fact binding the signing key, leading to contradiction and proving the `Cert_From_CA_Only` lemma.

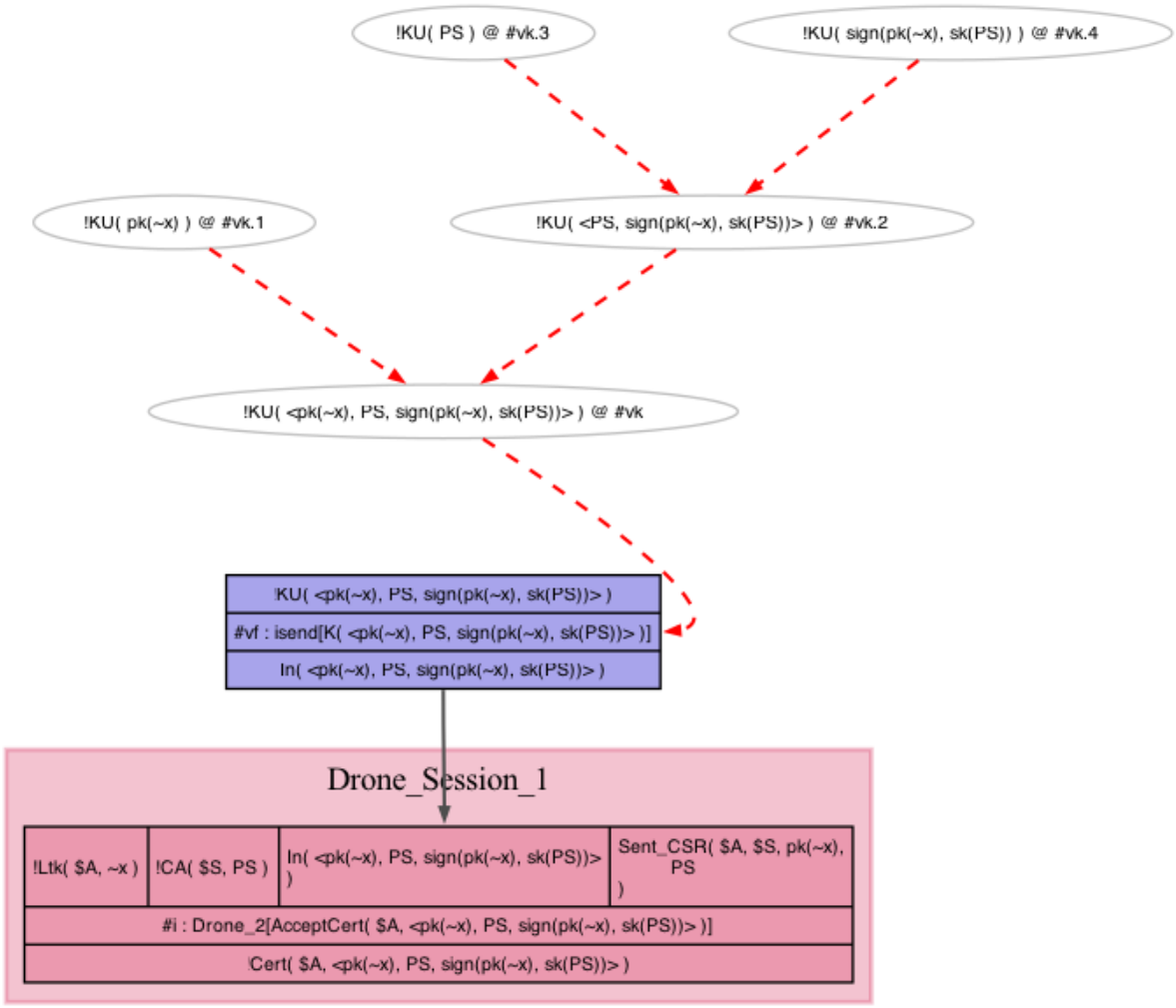


Figure 43: A valid trace showing a drone accepting a certificate `<PK, PS, SIG>` signed by a key `PS` associated with a trusted authority. This trace confirms that only legitimate CA-issued certificates are accepted.

To validate the integrity of issued authentication tokens (akin to JSON Web Tokens, JWTs), we proved the lemma `JWT_Authenticity`. This lemma ensures that any token used in the protocol `UseToken(TOKEN)` must have been issued `IssueToken(...)` by a legitimately trusted authentication server, and must be verifiably signed using the server's private signing key. This guards against token forgery, misuse, or substitution by adversaries.

The trace in Figure 44 confirms that every accepted token used in observer sessions was signed using a private key held by a known `TrustedAuthServer`. The red dashed inference arrows show that while the intruder may observe tokens and their usage in messages (e.g., as part of a signed RID query), they cannot generate a valid `TOKEN` unless it was previously issued by a trusted server.

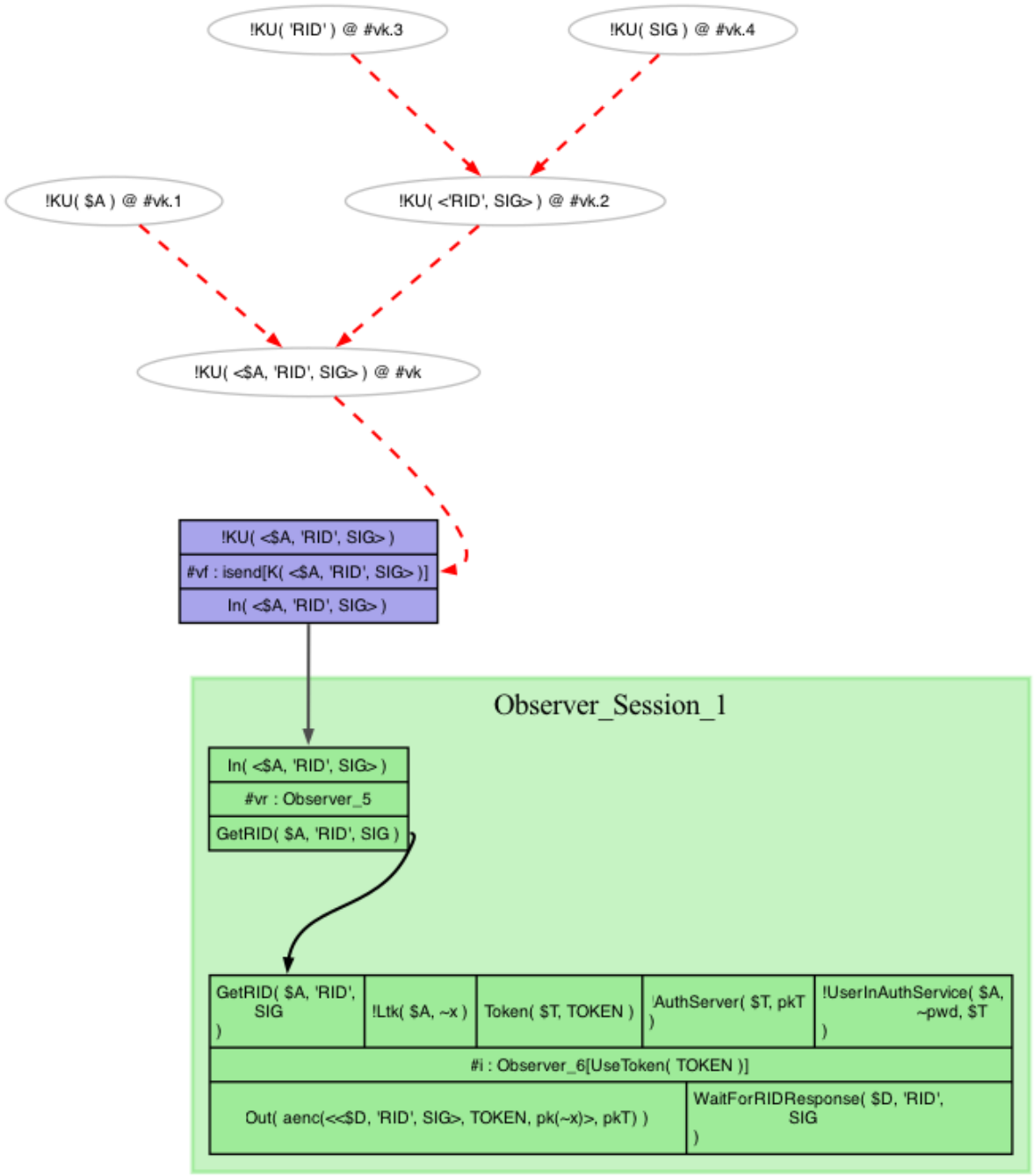


Figure 44: Tamarin trace proving the **JWT_Authenticity** lemma. The observer uses a token `TOKEN` that must have been signed by a legitimate authentication server. The intruder's inference paths (red dashed lines) cannot forge a valid token.

The `RID_Integrity` lemma ensures that every Remote ID message accepted by an observer has been properly signed by the entity that owns the corresponding public key. Specifically, it states that if a signed RID message `SIGRID` is accepted along with certificate `<PK, PS, SIG>`, then the signature must have been generated using the private key corresponding to the public key `pk(x)` embedded within the certificate.

Figure 45 presents a successful trace demonstrating the validity of this lemma. In the observer session, a RID message is received and accepted after successful cryptographic verification. The observer accepts the message only if the certificate and the signed RID match the internal consistency checks — ensuring that the public key in the certificate corresponds to the private key used to generate the signature over the RID.

The red dashed arrows illustrate the intruder's knowledge construction attempts. While the intruder learns the RID message, signature, certificate, and public keys, they are unable to produce an alternate `SIGRID` without access to the legitimate private key. The trace confirms that the `RIDAccepted(...)` event is triggered only when the signature `SIGRID` can be verified using a key derived from a valid certificate, which itself is signed by a trusted authority. Thus, the integrity of RID broadcasts is cryptographically guaranteed.

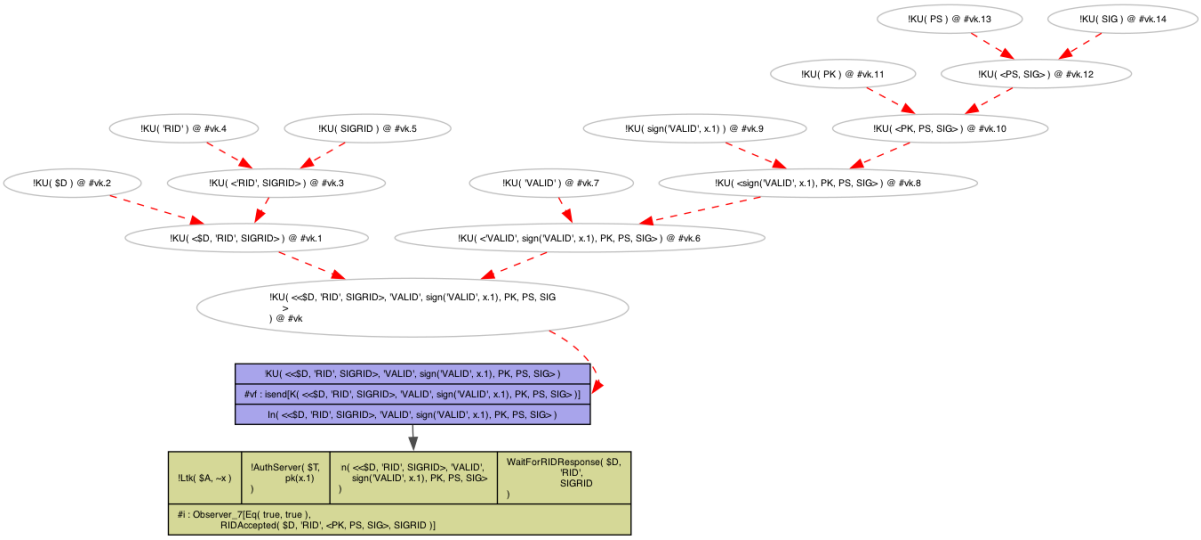


Figure 45: Tamarin proof trace confirming the **RID_Integrity** lemma in the AuthServer protocol. The observer accepts a signed RID message only after confirming that the signature `SIGRID` was created using the private key corresponding to the public key `PK`. The intruder's knowledge derivation paths show no ability to forge or spoof `SIGRID`.

The `Injective_Agreement` lemma aims to guarantee that each signed Remote ID message `SIG` is accepted for a single unique drone identity `D`. Formally, it states that if the same certificate `CERT` and signature `SIG` are accepted by any observer more than once, then those acceptance events must refer to the same point in the trace — meaning the messages originated from the same source, not from separate distinct identities.

However, the Tamarin Prover trace in Figure 46 presents a counterexample, thereby proving this lemma false. In this trace, an observer accepts the exact same `<D, 'RID', CERT, SIG>` tuple, but the two `RIDAccepted(...)` events occur at distinct traces`#i1` $\neq$ `#i2`, violating injectivity.

The root cause is that broadcast-based authentication protocols, even when layered with central server authentication and token usage, cannot prevent message replay. Once a signed RID message is broadcast and observed, any party — including an attacker — can rebroadcast the same message, leading the observer to accept the replayed signature as though it originated from the legitimate source.

The red dashed arrows in the trace illustrate how the attacker learns and replays the signed `<D, 'RID', SIG>` tuple. This results in two separate observers verifying and accepting the RID message independently, both believing the message to be valid and from the same source.

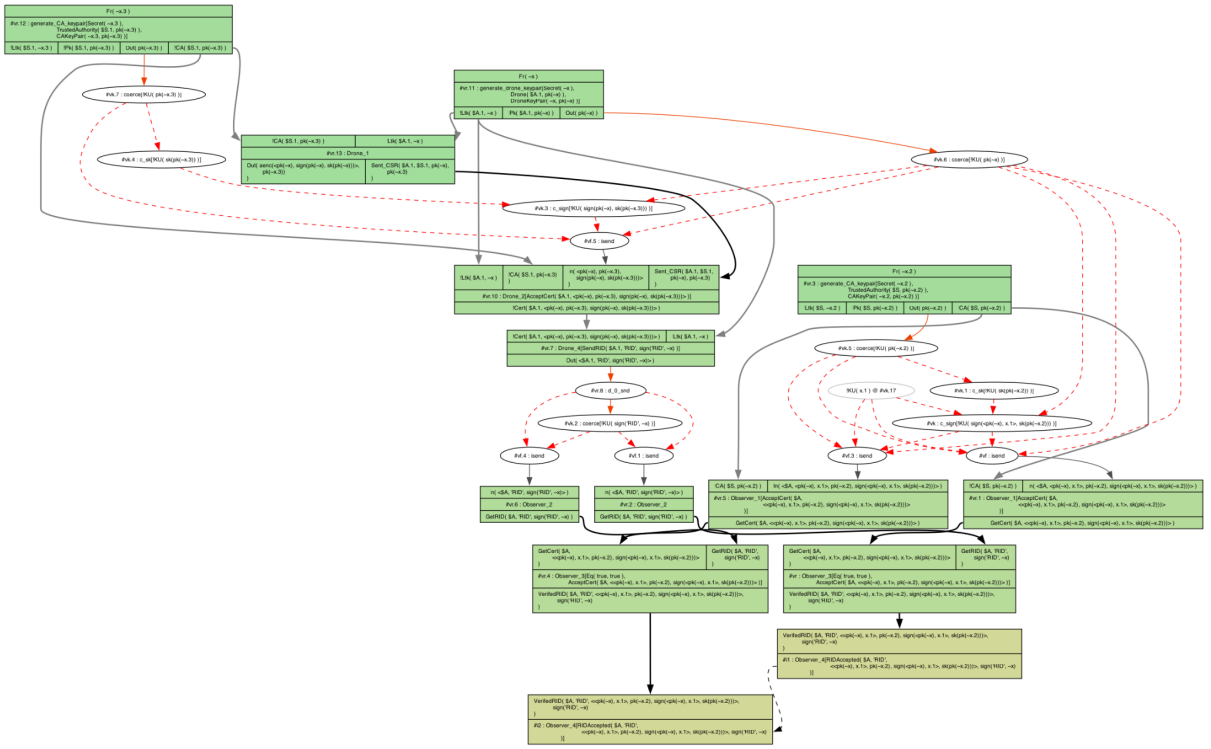


Figure 46: Tamarin Prover trace showing a counterexample to the **Injective_Agreement** lemma in the AuthServer-based Remote ID protocol. The same signed RID message is accepted by two observers from two different sources, demonstrating that message replays are possible and not cryptographically distinguishable.

5 Summary of Formal Verification Results

Table 1: Summary of Formal Verification Results

#	Lemma	Serverless Protocol	Authentication Server Protocol	Security Property
1	Protocol Liveness	**Verified**	**Verified**	Liveness
2	Private Key Secrecy	**Verified**	**Verified**	Secrecy
3	Certificate Integrity	**Verified**	**Verified**	Certificate Authenticity and Integrity
4	JWT Authenticity	**Not Applicable**	**Verified**	JWT Authenticity and Integrity
5	RID Message Integrity	**Verified**	**Verified**	RID Integrity
6	Injective Agreement	**Not Verified**	**Not Verified**	RID Integrity and Authenticity

Table 1 presents the results obtained from the Tamarin verification tool, which confirm that both of our protocol variants satisfy most of our desired core security properties. However, injective agreement could not be verified in either variant. This limitation arises because RID messages are bound to a drone's cryptographic identity but not to its physical source. As a result, a malicious drone can copy and rebroadcast the signed messages of a legitimate drone. If both drones broadcast concurrently, observers have no cryptographic means of distinguishing which drone actually generated the message. This enables adversaries to impersonate legitimate drones simply by replicating their RID broadcasts and mimicking their flight behavior. To ensure injective agreement within our protocol, further research is required into drone disambiguation techniques that can link broadcast messages to their true physical source.