Formal Verification of Permission Voucher Protocol

Khan Reaz and Gerhard Wunder
Cybersecurity and AI Research Group, Freie Universität Berlin
Emails: {khan.reaz, g.wunder}@fu-berlin.de

arXiv:2412.16224v1 [cs.CR] 18 Dec 2024

Abstract

The *Permission Voucher* protocol enables privacy-preserving and secure authentication using digital ID cards in smart-city infrastructure. This paper provides a formal analysis of the protocol, outlining its design, security guarantees, and potential vulnerabilities. Using tools like the Tamarin Prover, we formally verify critical security properties, including authentication, confidentiality, integrity, and replay prevention. This work highlights the effectiveness of formal verification in ensuring robust security for modern authentication systems.

I. Formal analysis

Formal analysis is about building a detailed mathematical model that shows how the protocol, its users, and any attackers behave. The model includes assumptions about the communication environment, the abilities of adversaries, and the goals of the protocol. By applying logical reasoning and mathematical proof techniques, it becomes possible to verify whether the protocol satisfies these security properties or identify conditions where the protocol may fail.

The key objective of formal analysis is to prove that certain desired security properties hold within the model. These properties, including confidentiality, integrity, authentication, and non-repudiation, are formally defined as logical assertions that the protocol must satisfy. For instance, a confidentiality property might be expressed as, *an adversary cannot deduce the contents of a message that was exchanged between two parties*. Formal methods allow for the rigorous testing of whether these properties are preserved under *all* conditions.

Three prominent approaches within this field are *process algebra, pi calculus, and symbolic models*. Each offers a unique perspective and set of tools for representing and reasoning about protocol interactions.

A. Process Algebra

Process algebra provides a framework for describing the behavior of concurrent systems, including security protocols. It represents processes as algebraic expressions and defines a set of operators for combining and manipulating these expressions. In the context of security protocols, process algebra can model the message exchanges between protocol participants, their internal state transitions, and the possible interleavings of actions.

By specifying the protocol's behavior as a process algebraic expression, we can reason about its properties using algebraic laws and equivalences. This allows us to prove or disprove various security properties, such as confidentiality, authentication, and integrity, by demonstrating that the protocol's behavior satisfies certain algebraic specifications.

B. Pi Calculus

Pi calculus is a process calculus specifically designed for modeling concurrent systems with mobile processes, making it well-suited for analyzing security protocols where communication channels and data can be dynamically created and passed around. In pi calculus, processes are represented as terms that interact through channels, exchanging messages and performing internal computations.

The applied pi calculus extends the basic pi calculus with constructs for modeling cryptographic operations, such as encryption, decryption, and digital signatures. This allows for the representation of security protocols and their potential interactions with an adversary. Verification tools like ProVerif leverage the applied pi calculus to automatically analyze protocols and prove or disprove their security properties.

C. Symbolic Models

Symbolic models provide a more abstract representation of security protocols, focusing on the symbolic manipulation of messages and their components rather than the underlying bit-level details. They typically represent messages as terms built from symbolic constants, variables, and function symbols representing cryptographic operations.

Tool	Type	Strengths	Shortcomings
Isabelle	Theorem Prover	High expressiveness for proofs in a wide range of domains, including cryptography.	Lacks built-in cryptographic primitives.
Coq	Theorem Prover	Powerful and general-purpose; can formally verify any kind of system, including cryptographic protocols	High learning curve, requiring expert knowledge of proof tactics; time-consuming for complex protocols
CryptoVerif	Computational Model	Computationally sound proofs, supports cryptographic primitives.	Complexity of tool usage; limited automation for large protocols.
Scyther	Symbolic Model	Fast unbounded verification, falsification, and protocol analysis	Does not support computational model; limited to symbolic verification of cryptographic protocols.
AVISPA	Model Checker	Automated validation of Internet security protocols, efficient model-checking for various security goals	Focused mainly on Internet protocols, less flexibility with user-defined cryptographic protocols
ProVerif	Symbolic Model	Extensive automation, can handle complex protocols, supports secrecy, authentication, and process equivalence properties.	Poor performance with large-scale protocols.
Tamarin	Symbolic Model	Combines symbolic analysis with explicit reasoning about cryptographic primitives; supports both bounded and unbounded protocol analysis. Active community and support.	Needs to master term/rules rewriting schema.

TABLE I: Summary of protocol verification tools

The behavior of the protocol is defined by a set of rules that govern how messages are constructed, sent, received, and transformed. Adversary capabilities are also modeled symbolically, allowing for the analysis of the protocol's resilience against various attack scenarios. Tools like Tamarin Prover utilize symbolic models to explore the state space of the protocol and verify or falsify desired security properties.

II. Verification tools

A wide range of security protocol verification tools (and languages) have been developed, each utilising distinct methodologies and offering unique capabilities for the analysis of security protocols. We provide an overview of some of the most well-known tools in ascending order below:

Isabelle/HOL, one of the oldest generic interactive theorem prover that requires manual input from a user in order to provide a high level of expressiveness and flexibility for formalising and verifying security protocols [7].

Coq is a proof assistant that enables the expression of mathematical definitions, executable algorithms, and theorems. It provides an environment for semi-interactive development of machine-checked proofs, ensuring their correctness and reliability [9].

CryptoVerif analyzes protocols in the computational model of cryptography. It employs game-based proofs to establish security guarantees within the model [1].

The **Scyther** model checker is well known for its user-friendliness and efficiency. By automatically exploring the state space of a protocol, Scyther can rapidly identify potential attacks and generate concrete attack traces [4].

The **AVISPA** framework integrates a number of protocol analysis tools, thereby facilitating the verification process. For example Coq can be used as a back-end with AVISA's high-level language. It supports both bounded and unbounded verification techniques [10].

ProVerif is based on the principles of applied pi calculus. It is able to confirm or disprove complex security properties, even when faced with the actions of a sophisticated adversary [2].

The **Tamarin Prover** is an effective tool for both bounded and unbounded verification, which involves proving the absence of attacks and falsification, or the identification of attacks. It employs a symbolic model augmented with temporal logic to express and verify intricate security properties that involve temporal ordering and causality [6].

III. Dolev-Yao adversary model

In 1983, Danny Dolev and Andrew Yao introduced a foundational adversary model for analysing security protocols in the fields of computer science and cryptography [5]. This model encapsulates *three* fundamental assumptions:

– The initial assumption, ***Perfect Cryptography***, presents a somewhat idealized view of cryptographic primitives, treating them as if they were black boxes within a symbolic model. This abstraction permits the representation of cryptographic operations as function symbols within an algebra of terms, complete with defining equations. Messages can be conceptualized as algebraic terms constructed from these primitives, providing a clean and abstract method for reasoning about protocol behavior without delving into the intricacies of specific cryptographic implementations. This assumption effectively circumvents the complexities of real-world cryptographic vulnerabilities, focusing instead on the logical structure and flow of protocols.

– The second assumption, called ***Unbounded Execution***, adds another way to scale and generalize the model. This means that a protocol can be run as many times as needed by any number of agents, who can take on different roles. This is important for understanding how protocols work in the real world, where they are run many times at once on large networks with lots of people involved. It allows us to look at how different combinations of runs affect the way the protocol works.

– The third and most distinctive assumption is the concept of a ***Network Adversary***. The adversary controls the communication network and can intercept, block, replay, and spoof messages. The adversary can also make messages look like they came from anywhere and put them into the network. This model shows a worst-case scenario of network security, where communication may be vulnerable to tampering or eavesdropping. The only thing this adversary can't do is break cryptographic protections. It can only learn the contents of messages that aren't secured or for which it doesn't have the keys.

IV. Why Tamarin prover?

We chose the Tamarin Prover for its proven capability to rigorously analyze the security of some of the real-world cryptographic protocols. In particular, Tamarin was used to propose solution to countering the Wi-Fi *KRACK* attack [3].

One of the main advantages of Tamarin is its multiset rewriting rule-based approach. This method allows us to describe how messages are exchanged, processed, and manipulated, not only by legitimate participants but also by potential adversaries. Tamarin's symbolic modeling makes it easier to capture a wide range of protocol behaviors and test them for vulnerabilities. By simulating different attack scenarios, Tamarin can automatically prove security properties, such as authentication and confidentiality, or identify specific attacks that threaten those properties.

Additionally, Tamarin offers both automated and interactive modes for proof construction. This versatility is invaluable for our work. The automated mode is efficient for routine analysis, quickly verifying protocol security or identifying flaws. However, when dealing with more complex protocols or challenging verification tasks, the interactive mode allows us to manually guide the proof process, investigate potential attacks in depth, and refine the analysis. This balance between automation and manual control gives us the flexibility needed to explore both simple and sophisticated security challenges.

Another key advantage is Tamarin's support for equational theories, which is crucial for protocols that rely on operations like Diffie-Hellman key exchange. This capability ensures that Tamarin can handle the advanced cryptographic techniques used in modern security protocols, making it especially useful for analyzing widely deployed systems such as Transport Layer Security (TLS) and Public Key Infrastructure (PKI).

Finally, Tamarin's track record in a variety of high-profile protocols such as `5G`, `TLS 1.2/1.3`, and `BLE` gave us the confidence to use it for the protocols we were developing.

V. Modeling Protocols with Tamarin Prover

To verify a security protocol using the Tamarin prover, a formal script must be constructed according to Tamarin's predefined syntax and conventions. This script is written in Tamarin's modeling language, which specifies the protocol's structure, roles, and security properties in a way that Tamarin can interpret and analyze. Each script must have the file extension `.spthy`, which stands for *Security Protocol Theory*.

A. Structure of a Tamarin Model

The structure of a Tamarin model typically consists of the following key components:

Declarations: The script begins with declarations of basic components such as messages, functions, and constants. These declarations define the basic cryptographic operations (e.g., encryption, decryption, hashing) and protocol-specific constants

that will be used throughout the model. For example: `functions: enc/2, dec/2, hash/1`

Here, `enc/2` and `dec/2` represent encryption and decryption functions with two arguments, while `hash/1` represents a hash function with one argument.

Rules: Protocol behavior is modeled using rules, which describe the flow of messages and actions between different entities (roles) in the system. Each rule consists of a set of premises (conditions that must be met) and consequences (actions that result if the conditions hold). Rules capture the core interactions of the protocol, such as sending and receiving messages, performing cryptographic operations, and updating the state of the system.

For example, a simple rule might represent a message being sent by one party and received by another:

```
    rule SendMessage:
    [ Fr(~msg), !SenderRole ]--[ msg_sent(~msg) ]->[ Out(~msg) ]
```

State Facts and Events: Tamarin uses facts to represent both static and dynamic information. Facts can be persistent (remaining true throughout the protocol execution) or ephemeral (holding temporarily). State facts track the current state of each participant and the protocol's progress. Events are used to record significant occurrences, such as when a message is sent or a particular security condition is met. These events can be used later to define and prove security properties.

Adversary Model: The adversary's capabilities are explicitly modeled in Tamarin scripts. By default, Tamarin assumes a Dolev-Yao adversary, meaning the attacker can intercept, modify, and inject messages into the network. The adversary model can be extended to include additional capabilities, such as controlling certain protocol participants or interacting with specific cryptographic operations.

Adversary actions are typically represented using rules that define how the adversary can manipulate messages or participate in the protocol. For example:

```
rule AdversaryIntercept:
    [ In(~msg) ] --[ ]-> [ Out(~msg), !Adversary ]
```

Security Properties (Lemmas): The final section of the script defines the security properties to be verified, expressed as lemmas. These lemmas state the conditions that should hold true for the protocol to be secure. Examples include secrecy (confidentiality of a message), authentication (verifying the identity of participants), and integrity (ensuring that messages are not tampered with).

To illustrate how the Tamarin Prover works, let's consider examples for symmetric encryption, asymmetric encryption, message integrity, and authentication.

In *symmetric encryption*, both parties share a secret key. To model this, define a function `enc/2` for encryption and `dec/2` for decryption, along with an equation like `dec(enc(m, k), k) = m`. A rule might simulate a sender encrypting a message and a receiver decrypting it. The lemma would ensure that only the intended recipient can decrypt the message, verifying *confidentiality*.

For *asymmetric encryption*, define functions `aenc/2` for encryption and `adec/2` for decryption, with `pk/1` representing the public key. The equation `adec(aenc(m, pk(k)), k) = m` models decryption using the private key. A rule would simulate a sender encrypting a message with the receiver's public key. The lemma ensures that only the receiver, with the corresponding private key, can decrypt it, verifying confidentiality and authenticity.

To ensure message *integrity*, use a function `mac/1` for message authentication codes. A rule might model a sender generating a MAC for a message and a receiver verifying it. The lemma checks that if the MAC is valid, the message has not been altered, ensuring integrity.

For *authentication*, define a challenge-response protocol. A rule might simulate an authenticator sending a challenge and the enrollee responding with a computed value.

B. Verification Process

Once the protocol is modeled, Tamarin then uses its automated reasoning engine to either prove or disprove these lemmas based on the constructed model:

Proven Lemmas: If a lemma is successfully proven, it means the protocol satisfies the corresponding security property within the defined adversary model. This guarantees that the protocol upholds critical security attributes, such as confidentiality or authentication.

Falsified Lemmas: If Tamarin fails to prove a lemma, it produces a counterexample trace. This trace provides a step-by-step sequence of actions or messages that demonstrate how an attacker could exploit the protocol, revealing potential vulnerabilities or weaknesses. These counterexamples are valuable in diagnosing flaws, as they allow protocol designers to understand and rectify potential attacks.

VI. Visual Meaning of the Dependency Graphs

The following explanation of the Tamarin dependecny graph is extracted from [8].

If a lemma is successfully proven, it is highlighted in green; if a counterexample is found, it appears in red. Various arrow types and colors denote different elements in the proof visualization:

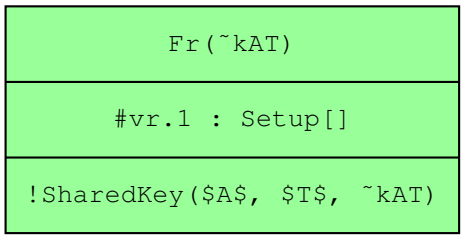
Black arrows: Indicate where a produced fact is utilized by another rule.

Grey arrows: Indicate the use of persistent facts or adversary knowledge by another rule.

Red arrows: Indicate when the adversary obtains a term from the network via an `Out` fact.

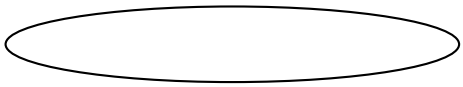
Dotted arrows: Represent the temporal ordering or precedence of actions.

Green rectangular boxes: Rule instances are depicted following a specific format. For a rule represented as `LHS--[ACT]--> RHS`, the top line corresponds to the LHS (left-hand side), the middle line denotes the ACT (action), and the bottom line represents the RHS (right-hand side). Slight variations in shades of green are employed to distinguish between instances of different rules. The action line within the box specifies the timepoint variable for a given rule instance, along with the rule name and any associated set of actions.



Grey ellipses: Indicate actions where the adversary selects a term, such as `t1`. The term `!KU` represents a specific case of `K`, signifying the adversary's acquisition of a term.

Black ellipses: Denote adversary actions, such as deriving knowledge or transmitting messages.



TECHNICAL INTERNALS

`isend`: Represents an adversary sending a message to an `In` fact.
`!KU`: Terms used during the construction path.
`coerce`: Indicates the start of the construction path after any potential deconstruction path.

VII. DEMONSTRATING A REPLAY ATTACK IN TAMARIN

In this example, we model a scenario where an attacker can replay a message, bypassing the security mechanism. This is a common issue in protocols without proper checks for uniqueness or freshness of messages.

```
theory ReplayAttack
begin

functions: mac/2

// Key registration for message integrity
rule Register_Key:
  [ Fr(~k) ] --> [ !MacKey($A, ~k) ]

// Client sends message with a MAC (no freshness check)
rule Client_Sends_Message:
  let m = mac('message', ~k)
  in
  [ !MacKey($A, ~k) ] --> [ Out(m) ]

// Server receives the message and checks MAC (no nonce or freshness)
rule Server_Receives_Message:
  let m = mac('message', ~k)
  in
  [ In(m), !MacKey($A, ~k) ] --> [ Out('message_verified') ]

// Lemma to ensure replay attack is possible
lemma Replay_Possible:
  "Ex m #i #j.
   (In(m) @ #i & In(m) @ #j & #i < #j)"

end
```

In the above example, the server simply checks if the received message has a valid MAC, without verifying that the message is fresh (i.e., not a replay of an old valid message).

Lemma `Replay_Possible` checks if the same message `(m)` can be received multiple times, indicating a replay attack.

Running `tamarin-prover ReplayAttack.spthy --prove` will show the lemma `Replay_Possible` is **falsified**, meaning Tamarin detects the possibility of a replay attack. Specifically, the output will include a trace showing that the same message was sent and verified multiple times without any checks for freshness, which indicates the success of a replay attack.

Lemma (Replay Possible)**.**

$$\exists m\#i\#j.\ ((In(m)\ @\#i) \wedge (In(m)\ @\#j)) \wedge (\#i < \#j)$$

Guarded formula characterizing all counter-examples:

$$\forall m\#i\#j.\ (In(m)\ @\#i) \wedge (In(m)\ @\#j) \Rightarrow \neg(\#i < \#j)$$

Now, we introduce a nonce (a unique value used only once) to prevent the replay attack. Each message will include a nonce, and the server will only accept fresh, non-repeated messages.

```
theory PreventReplayAttack
begin

functions: mac/2
```

```
// Key registration for message integrity
rule Register_Key:
  [ Fr(~k) ] --> [ !MacKey($A, ~k) ]

// Client sends a message with a MAC, including a fresh nonce
rule Client_Sends_Message:
  [ Fr(~n) ]  // Generate a fresh nonce
  let m = mac('message', ~k)
  in
  [ !MacKey($A, ~k) ]
  --> [ Out(<~n, m>), !Nonce(~n) ]

// Server checks nonce and verifies message
rule Server_Receives_Message:
  let m = mac('message', ~k)
  in
  [ In(<n, m>), !MacKey($A, ~k), not(Nonce(n)) ]
  --> [ Out('message_verified'), !Nonce(n) ]

// Ensure replay attack is not possible due to nonce check
lemma No_Replay_Attack:
  "All n m #i #j.
   (In(<n, m>) @ #i & In(<n, m>) @ #j) ==> #i = #j"

end
```

Running `tamarin-prover PreventReplayAttack.spthy --prove` will verify that no replay attacks are possible. The lemma `No_Replay_Attack` will be verified, indicating that the nonce mechanism successfully prevents the same message from being accepted multiple times.

VIII. Formally Verified Security Properties

The following security properties are considered while evaluating each protocols with the Tamarin prover.

Authentication is the process of verifying the identity of a user or device before granting access to resources or services. It ensures that only authorized entities can interact with the system, reducing the risk of unauthorized access. Authentication mechanisms can range from simple password-based systems to multi-factor approaches involving biometrics, cryptographic tokens, or one-time passwords. The robustness of authentication directly impacts the overall security of the system, as weak or compromised authentication can lead to breaches.

Mutual authentication extends the concept of authentication by requiring both parties in a communication to verify each other's identities. This process ensures that both the client and the server (or two devices) are genuine and trustworthy before data is exchanged. Mutual authentication is crucial in preventing impersonation attacks, where one party might otherwise be deceived by a malicious actor posing as the other. Common methods for achieving mutual authentication include TLS with client certificates or cryptographic challenge-response protocols.

A **Man-in-the-Middle (MitM)** attack occurs when an attacker intercepts communication between two parties without their knowledge, allowing the attacker to eavesdrop or modify the transmitted data. MitM attacks exploit the lack of secure communication channels and can lead to the leakage of sensitive information, identity theft, or the injection of malicious content. Strong encryption, mutual authentication, and secure key exchange protocols are commonly used to defend against MitM attacks by ensuring that attackers cannot decrypt or tamper with messages.

Key confirmation is a process that ensures both parties in a communication have agreed upon and are using the same cryptographic key before secure communication can proceed. This verification step guarantees that the key exchange was successful and that no third party has tampered with or intercepted the key. Key confirmation adds an extra layer of security to key exchange protocols, ensuring that data encrypted with the agreed-upon key remains confidential and secure.

Uniqueness of the session ensures that each communication session between parties is distinct and isolated from others, preventing session reuse or replay attacks. This property is typically achieved through the use of unique session identifiers or nonces, ensuring that even if an attacker intercepts session information, they cannot replicate or hijack the session. Ensuring session uniqueness is crucial in maintaining the integrity and confidentiality of data in repeated interactions between parties.

Non-repudiation ensures that a party involved in a communication or transaction cannot later deny their involvement. This property is achieved through cryptographic techniques like digital signatures, which provide verifiable proof that a specific party sent a particular message or initiated a transaction. Non-repudiation is especially important in legal and financial contexts, where accountability and proof of action are critical for trust and compliance.

Confidentiality refers to the protection of data from unauthorized access, ensuring that only intended recipients can read the transmitted information. This is typically achieved through encryption, where data is transformed into a format that is unreadable to anyone without the proper decryption key. Confidentiality is a fundamental aspect of information security, safeguarding sensitive data from eavesdropping and preserving privacy in both communication and data storage.

Integrity ensures that the data being communicated or stored remains unaltered and accurate throughout its lifecycle. It guarantees that any unauthorized changes to the data, whether intentional or accidental, can be detected. Integrity is typically enforced through cryptographic hash functions, checksums, or message authentication codes (MACs), which allow the receiver to verify that the data has not been tampered with. Maintaining data integrity is critical in preventing data corruption and ensuring trustworthiness.

Replay prevention is a security measure that protects systems against attacks where an adversary captures and retransmits valid data or authentication credentials in an attempt to gain unauthorized access. By using unique session identifiers, timestamps, or nonces in each communication, replay prevention ensures that previously captured messages cannot be reused in a fraudulent way. This mechanism is essential for maintaining the integrity of authentication and transaction processes.

Key validation ensures that a cryptographic key used in communication or encryption is valid, secure, and not compromised. Before initiating secure communication, both parties verify the authenticity and correctness of the key to confirm that it hasn't been tampered with or substituted by an attacker. Proper key validation prevents the use of weak, expired, or maliciously altered keys, maintaining the overall security of the cryptographic system.

Remark. When presenting the protocol in the following sections, we include all the lemmas that are modeled and verified. Additionally, where appropriate, we provide dependency graphs to illustrate the relationships among these lemmas.

IX. Formal Verification of **Permission Voucher**

To prove the *PermissionVoucher* protocol in Tamarin, we need to focus on different phases of the protocol, their respective security properties, and the associated risks. Here, we define trust relationships and channels, then we identify key security lemmas.

A. Trust Relationships and Channels

Secure Channels:

Channel between ID APP and ID-CARD (NFC):
Denoted as H_{IC}, this channel is assumed secure as it relies on physical NFC communication, involving local proximity. Confidentiality and integrity are guaranteed since the channel is established via NFC and PIN verification.

Channel between ID APP and OWNER (PIN Input):
H_{PIN} is also assumed secure due to user input of a PIN, which only the user knows. Authentication of the OWNER is provided via PIN.

Public or Partially Trusted Channels:

Channel between OWNER and On-premise SERVICE (UWB connection):
H_{UWB} is considered partially trusted, as it may provide local network access via UWB, but the security may depend on physical constraints (proximity). Messages over this channel may be intercepted, and hence the protocol must prevent impersonation and replay attacks.

Channel between On-premise SERVICE and ID-VERIFIER:
H_{SI} is partially trusted. It involves verification of data provided by the On-premise SERVICE using the ID-VERIFIER. The integrity of the transmitted data is crucial, and this channel may be vulnerable to attacks like tampering.

B. Security lemmas for Permission Voucher

Lemma (Authentication: Ensure that only a valid ID-CARD can be unlocked by a correct PIN)**.**

$$\forall pin\, card\_data\, \#i\, \#j.\ ((In(pin\_input(pin))\ @\ \#i) \wedge (In(nfc\_connection(card\_data))\ @\ \#j)) \Rightarrow (\#i < \#j)$$

Guarded formula characterizing all counter-examples:

$$\exists pin\, card\_data\, \#i\, \#j.\ (In(pin\_input(pin))\ @\ \#i) \wedge (In(nfc\_connection(card\_data))\ @\ \#j) \wedge \neg(\#i < \#j)$$

Lemma (Voucher Authenticity: Ensure that only the ID APP can create a valid permission voucher)**.**

$$\forall voucherID\, data\_items\, private\_key\, \#i\, \#j.\ ((Out(sign(create\_voucher(voucherID, data\_items), private\_key))\ @\ \#i) \wedge (K(private\_key)\ @\ \#j)) \Rightarrow (\#i = \#j)$$

Guarded formula characterizing all counter-examples:

$$\exists voucherID\, data\_items\, private\_key\, \#i\, \#j.\ (Out(sign(create\_voucher(voucherID, data\_items), private\_key))\ @\ \#i) \wedge (K(private\_key)\ @\ \#j) \wedge \neg(\#i = \#j)$$

Lemma (Voucher Integrity: Ensure that a permission voucher is not modified during transfer)**.**

$$\forall voucherID\, serviceID\, private\_key\, \#i\, \#j.\ ((Out(sign(redeem\_voucher(serviceID, voucherID), private\_key))\ @\ \#i) \wedge (In(sign(redeem\_voucher(serviceID, voucherID), private\_key))\ @\ \#j)) \Rightarrow (\#i = \#j)$$

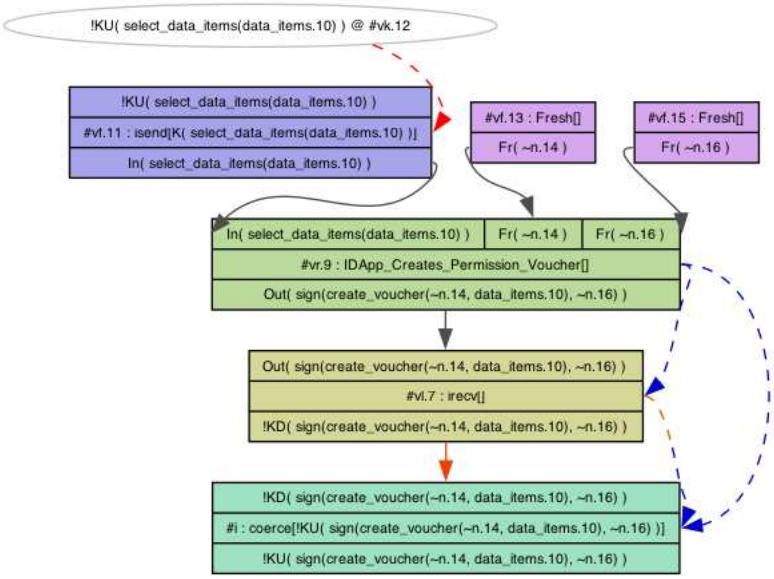


Fig. 1: Tamarin dependency graph when IDApp creates PermissionVoucher

Lemma (No Replay Nonce: Ensure that each nonce used during the process is unique and prevents replay attacks)**.**

$$\forall nonce\, \#i\, \#j.\ ((In(nonce(nonce))\ @\ \#i) \wedge (In(nonce(nonce))\ @\ \#j)) \Rightarrow (\#i = \#j)$$

Lemma (Visitor Pass Authenticity: Ensure that a visitor pass can only be generated by the ID-VERIFIER after a valid voucher is redeemed)**.**

$$\forall passID\, voucherID\, private\_key\, verifier\_key\, \#i\, \#j.\ ((In(sign(redeem\_voucher(serviceID, voucherID), private\_key))\ @\ \#i) \wedge (In(sign(generate\_visitor\_pass(passID), verifier\_key))\ @\ \#j)) \Rightarrow (\#i < \#j)$$

Guarded formula characterizing all counter-examples:

$$\exists passID\, voucherID\, private\_key\, verifier\_key\, \#i\, \#j.\ (In(sign(redeem\_voucher(serviceID, voucherID), private\_key))\ @\ \#i) \\ \wedge\ (In(sign(generate\_visitor\_pass(passID), verifier\_key))\ @\ \#j) \wedge \neg(\#i < \#j)$$

Lemma (Data Items Confidentiality: Ensure that selected data items remain confidential unless explicitly shared by the OWNER)**.**

$$\forall data\_items\, \#i\, \#j.\ ((Out(select\_data\_items(data\_items))\ @\ \#i) \wedge (K(data\_items)\ @\ \#j)) \\ \Rightarrow (\#i \\ = \#j)$$

Guarded formula characterizing all counter-examples:

$$\exists data\_items\, \#i\, \#j.\ (Out(select\_data\_items(data\_items))\ @\ \#i) \wedge (K(data\_items)\ @\ \#j) \wedge \neg(\#i = \#j)$$

Lemma (Mutual Authentication)**.**

$$\forall serviceID\, verifier\_key\, \#i\, \#j.\ ((In(sign(verify\_voucher(serviceID), verifier\_key))\ @\ \#i) \wedge (In(session\_id(serviceID))\ @\ \#j)) \\ \Rightarrow (\#i \\ < \#j)$$

Guarded formula characterizing all counter-examples:

$$\exists serviceID\, verifier\_key\, \#i\, \#j.\ (In(sign(verify\_voucher(serviceID), verifier\_key))\ @\ \#i) \\ \wedge\ (In(session\_id(serviceID))\ @\ \#j) \wedge \neg(\#i < \#j)$$

C. Tamarin output for Permission Voucher

Table II presents a summary of the output from the Tamarin prover for each security lemma.

Analyzed:	permissionvoucher.spthy
Processing Time:	0.25s
Authentication (all-traces):	verified (4 steps)
Voucher_Authenticity (all-traces):	verified (4 steps)
Visitor_Pass_Authenticity (all-traces):	verified (4 steps)
Data_Items_Confidentiality (all-traces):	verified (4 steps)
Mutual_Authentication (all-traces):	verified (4 steps)

TABLE II: Tamarin output summary for Permission Voucher

D. Identified weaknesses

The ID-VERIFIER is trusted to validate the voucher and generate the visitor pass, but it could be vulnerable if compromised. Adding additional security checks, such as mutual authentication between the On-premise SERVICE and ID-VERIFIER, would ensure both entities are verified before any sensitive information is exchanged.

The current model doesn't account for potential attacks involving compromised parties (e.g., On-premise SERVICE acting maliciously). Adding role-specific lemmas to analyze potential insider threats could strengthen the model. For instance, add a lemma to verify that the ID-VERIFIER is the only entity capable of authenticating the voucher, preventing other entities from replaying or impersonating the verification process. Ensure that the On-premise SERVICE is properly registered and bound to the verification process. This could prevent adversaries from pretending to be the On-premise SERVICE.

There is no current check for the expiry of permission vouchers. Implementing a mechanism for verifying the time validity of a voucher (e.g., using timestamps) would prevent expired vouchers from being reused by adversaries. A `Voucher_Expiry` lemma could ensure that any voucher presented for validation is within its valid time frame.

The model currently assumes that all keys (e.g., `private_key, verifier_key`) are securely managed and known to the respective parties. Introducing key distribution and management rules would add another layer of security to the protocol, reducing the risk of keys being compromised.

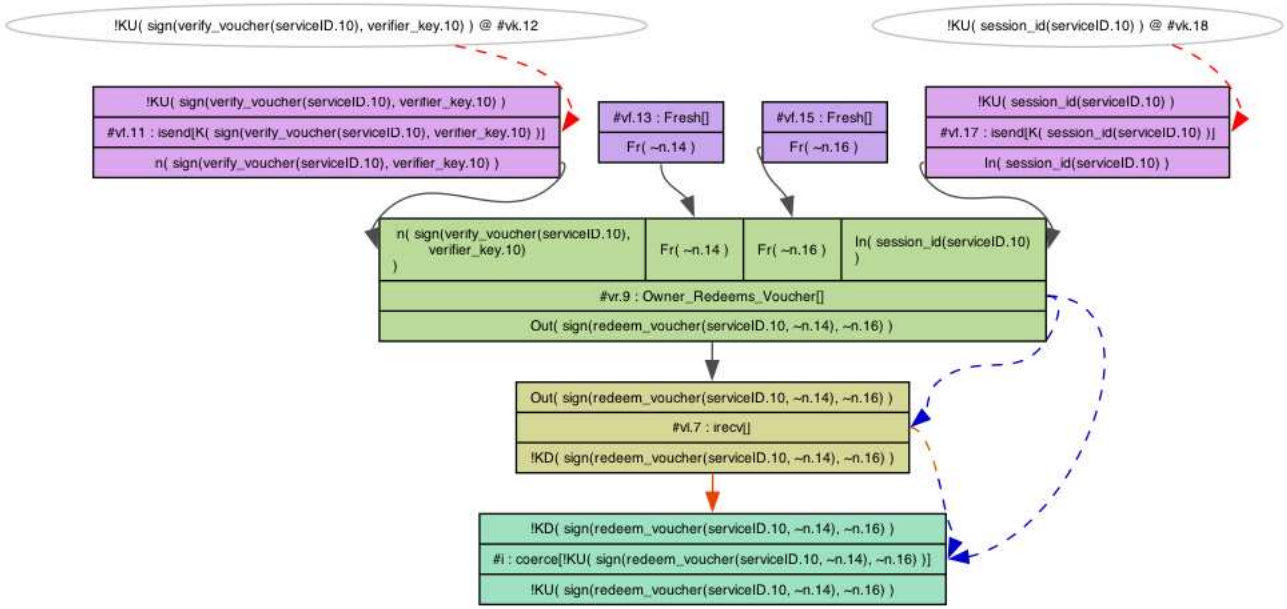


Fig. 2: Tamarin dependency graph when OWNER redeems voucher

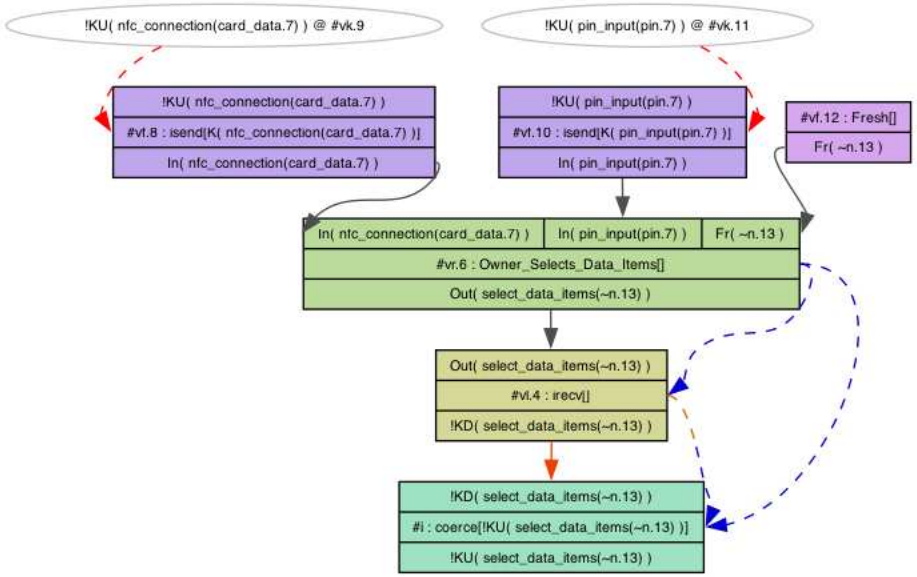


Fig. 3: Tamarin dependency graph when OWNER selects card data items

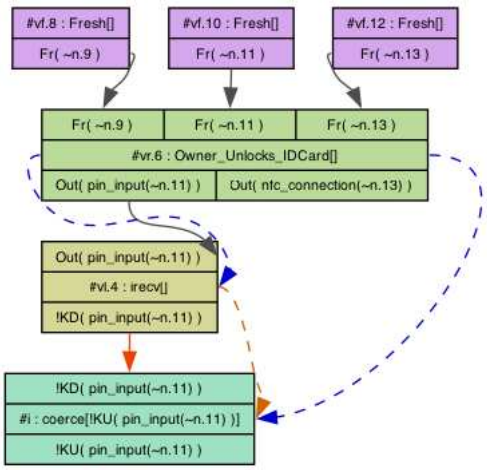


Fig. 4: Tamarin dependency graph when OWNER unlocks IDCard

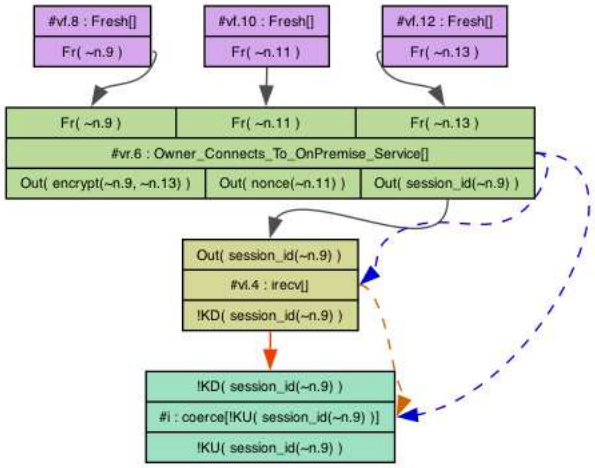


Fig. 5: Tamarin dependency graph when OWNER connects to OnPremise Service

X. Known Limitations

We note the following key limitations associated with using the Tamarin prover for verifying security protocols, as well as some current development challenges within the framework.

Proof Strategy. Tamarin operates within a symbolic model, whereby cryptographic operations are abstracted into symbolic terms, rather than being represented by fully detailed cryptographic functions. Consequently, Tamarin is only capable of detecting attacks that adhere to the aforementioned symbolic representation and a predefined set of equations, collectively known as subterm-convergent theories. Although it is capable of supporting some built-in functions, its capacity to handle more complex or non-standard cryptographic theories remains limited. Furthermore, due to the intrinsic complexity of the problems it addresses, Tamarin may not always complete its analysis, particularly for protocols with extensive or intricate state spaces. In such instances, it may be necessary to intervene manually by proving individual lemmas or diagnosing the reason for the failure of the trace analysis to terminate.

Trace Mode vs. Observational Equivalence Mode. Tamarin employs two principal modes of analysis: *trace mode* and *observational equivalence mode*. Trace mode is typically regarded as a reliable and comprehensive approach, capable of identifying all potential attack traces within the confines of the symbolic model. However, the observational equivalence mode, which aims to demonstrate that two distinct executions of a protocol are indistinguishable to an adversary, remains a work in progress. Although it is a reliable method to some extent, it may fail to identify certain attacks due to the necessity for precise alignment between the rules. Furthermore, observational equivalence mode does not yet fully support all protocol features, particularly restrictions or complex message structures, which can result in incomplete analysis.

Scalability and Performance. The complexity of formal verification represents a significant challenge for Tamarin, particularly in the context of large or state-heavy protocols, where the scalability of its analysis may be limited. In the case of protocols comprising a multitude of roles, message exchanges, or state transitions, the time required for Tamarin to complete its analysis may increase exponentially. Such limitations may result in performance bottlenecks and, in some instances, impede the prover's ability to reach a conclusion within a reasonable timeframe.

REFERENCES

[1] Bruno Blanchet. Cryptoverif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, volume 117, page 156, 2007.

[2] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial. *Version from*, 16:05–16, 2018.

[3] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. A formal analysis of {IEEE} 802.11's {WPA2}: Countering the kracks caused by cracking the counters. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1–17, 2020.

[4] Cas JF Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 119–128, 2008.

[5] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[6] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 696–701. Springer, 2013.

[7] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

[8] Tamarin Prover Team. Tamarin prover. https://github.com/tamarin-prover/tamarin-prover, 2024. Accessed: 2024-01-20.

[9] David Von Oheimb. The high-level protocol specification language hlpsl developed in the eu project avispa. In *Proceedings of APPSEM 2005 workshop*, pages 1–17. APPSEM'05, Tallinn, Estonia, 2005.

[10] David Von Oheimb. The high-level protocol specification language hlpsl developed in the eu project avispa. In *Proceedings of APPSEM 2005 workshop*, pages 1–17. APPSEM'05, Tallinn, Estonia, 2005.