
© Copyright 2009 [please consult the authors]
Modeling and Verification of Privacy Enhancing Security Protocols

Suriadi Suriadi, Chun Ou Yang, Jason Smith, Ernest Foo
Queensland University of Technology, Australia
{s.suriadi, c.ouyang, j4.smith, e.foo}@qut.edu.au

Abstract. Privacy enhancing security protocols (PESPs) are a family of protocols that allow secure exchange of sensitive user information. They are important in preserving users privacy in today’s open environment. Like other security systems, proof of the correctness of PESPs is necessary before they can be deployed. However, the traditional provable security approach, though well established for verifying cryptographic primitives, is not applicable to PESPs. We apply the formal method of Coloured Petri Nets (CPNs) to construct an executable specification of a representative PESP, namely the Private Information Escrow Bound to Multiple Conditions Protocol (PIEMCP). Formal semantics of the CPN specification allows us to reason about various security properties of PIEMCP using state space techniques. This investigation has also led us to a number of approaches for modeling and verification of PESPs in general, demonstrating the benefit of applying CPN-based formal approach to proving the correctness of security protocols.

1 Introduction

As a response to the increasing number of incidents compromising the privacy of millions of users [1–5], there has been an increase in the research related to privacy enhancing security protocol (PESP). A PESP is a generic term that refers to protocols whose main purpose is to preserve users privacy in an on-line environment; for example, emulating the off-line anonymity afforded by cash transaction, a PESP ensures that when a user purchases some goods on-line, the on-line seller (also known as a service provider (SP)) does not learn the identity of the user. Normally, PESPs apply existing cryptographic primitives (such as secret sharing and encryption techniques) to provide the privacy-enhancing features. In recent times, the Trusted Platform Module (TPM) technology - which provides secure hardware storage of cryptographic keys and an implementation of common cryptographic primitives - has also been used in PESP [6].

In this paper, we are interested in the modeling and verification of cryptographic-based PESPs. In the cryptography domain, the main approach used to verify a cryptographic primitive security is usually based on the provable security approach [7]. Unfortunately, these techniques cannot simply be used to verify security properties of PESPs. The main reasons are due to (1) simplified assumptions employed in the provable security approach are not realistic in the
PESP environment, and (2) the scalability problem when the variables to consider in assessing the security of a system goes beyond the relatively well-defined boundaries normally encountered in cryptography [8].

Figure 1 illustrates the layers of attacks and security properties involved in delivering a secure PESPs (and other systems that apply cryptographic primitives). We can see that the types of security properties that we are interested differ between layers. For example, for a signature scheme, we are interested in verifying if the signature scheme has achieved the unforgeability property.

At the application layer (in which PESP resides), we are interested in whether the orchestration of the cryptographic primitives employed deliver some application-specific properties. For example, in PESPs, we are interested in verifying if the claimed privacy behaviours of the protocol, such as whether the protocol provides a the revokable-but-abuse-resistant anonymity property. At this layer, the required security properties are application-specific, therefore, they vary greatly from one application to another.

Security properties are realized in the context of a set of attack models. As shown in Figure 1, as we move up the layers, the types of attacks that we need to consider also change. The attack models for cryptographic primitives are generally well-known with well-defined boundaries [8] (such as the random oracle model [9]). Similarly, a network attack model is well-defined, such as the Dolev-Yao model [10]. At the application layer, however, there is no ‘standard’ attack model that is valid across applications. For example, the attack models used for electronic cash systems are different from those used for PESPs due to the different entities, goals, and security technologies applied to them.

Finally, at the application layer, the complexity of the system has increased significantly as we now need to consider substantially more variables in the security assessment, such as the number of entities involved, the number and types of cryptographic primitives used, the number of message exchanges between the entities, the assumed infrastructure, and so on. Each of these factors could introduce new security vulnerabilities and exhaustive enumeration of all possible attacks is impossible [8].

Given the different security requirements and attack models between cryptographic primitives and applied cryptographic systems, such as PESPs, it is not difficult to see why the provable security approach is not suitable. The provable security approach reduces the security of a cryptographic system to some difficult hard (normally mathematical) problems within the context of some standard attack model with well-defined boundaries. Some security properties are said to have been achieved when it is shown that the properties depend on some hard problems which are not solvable in a polynomial time. This is not a suitable approach to PESPs because (1) the security properties involved in PESPs are such that it is counter-intuitive, if not misleading and impossible, to reduce them to some hard mathematical problems, and (2) while the provable security approach is well-established for verifying the security of cryptographic primitives, it is unable to scale to support PESP’s large, and growing, number of attacks. The introduction of a new attack requires all security properties to be manually
re-verified. There is no known tool to automate the provable security verification process. In summary, the provable security approach is necessary, but not sufficient for the modeling and verification of PESPs due to the different security dimensions of a system they are used.

Proposed Approach Given the reasoning just detailed, we require an approach that can (1) model a PESP such that its behaviours, including the behaviours of the cryptographic primitives used, can be captured, and (2) be easily configurable to capture various types of attacks so that when new types of attacks are discovered, we could easily extend the model with this new information and automatically re-evaluate the protocol to verify if the security properties still hold. Formal method approach not only supports these features but also provides many additional benefits - details followed.

In this paper we propose an approach to modeling and verification of one PESPs - the Private Information Escrow Bound to Multiple Conditions (PIEMC) protocol [11] - using the Coloured Petri Net (CPN) technique [12] with the help of the CPN Tool [13].

We choose to use CPN Tool over other formal methods because (1) it provides the ability to model and capture the behavioural properties of a system, especially the concurrent behaviours, (2) it provides graphical interface that allows easy understanding of the protocol being modeled; consequently, it allows easy protocol debugging and allows PESP experts (who may not necessarily be experts in CPN) to validate if the model is a faithful representation of the PESP, (3) it has a wide-range of existing analysis techniques, including state-space analysis and state-invariants, (4) it supports modular approach to specifying complex PESP, (5) it supports flexible data type definition which is very useful in representing cryptographic data, (6) CPN Tool is integrated with Standard
ML (SML) inscription language [14] which proves to be very useful in capturing many cryptographic processing behaviours, and (7) it allows attack models to be parameterized such that various attack behaviours can be added to the model incrementally, and consequently, allows re-use of the already-built artifacts required for verifying the security properties of the protocol. In addition, because CPN has also been commonly used to analyse a system’s performance, and because efficiency is a major concern in PESP due to the use of resource-intensive cryptographic primitive, it is a logical step to model and verify PESP using CPN which can be later extended for performance analysis as well.

Our approach to using CPN in the modeling and verification of PESP consists of several dimensions. In terms of modeling, a separation of concerns is applied between the message-flow of PESPs from the processing (including cryptographic processing) of those messages: the graphical interface of CPN is useful for capturing the PESPs message flow, especially the concurrent behaviours, while the integrated CPN inscription language (the Standard ML) is very powerful in capturing a wide variety of cryptographic primitives and their associated processing behaviours. In addition, the integrated textual input-output operation supported in SML proves to be very useful in the processing of PESPs session data.

When verifying the security properties of a PESP, we use two approaches: (1) querying and analysing the stored session data at each of the entities involved in the protocol (session-data analysis), and (2) translating a set of common PESP security properties into a series of statements whose correctness can be verified using standard state-space analysis.

We identify repeatable patterns in both the modeling and the verification dimension of our approach. Based on these identified patterns, we propose a set of guidelines, notably guidelines related to how we could capture the behaviour of cryptographic primitives in CPN) that can be used to model and verify PESPs and other applied cryptographic protocols.

Furthermore, the case-study protocol used in this paper also employs the TPM technology, in particular, those features of TPM technologies that preserve user’s privacy [15] and provide the provable execution property [16]. These features of TPM are very useful in PESP due to their ability to massively improve the efficiency of PESPs: it allows one to remove many of the inefficient resource-intensive cryptographic operations commonly used in PESPs. Therefore, we expect the modeling guidelines to capture those TPM behaviours that we present in this paper to be applicable to a wide-range of PESPs as well as other systems that rely on the privacy-preserving and provable execution features of TPMs.

To summarize, there are two main contributions detailed in this paper: the proposal of using CPN and CPN Tool for the purpose of modeling and verifying security properties of PESPs using the PIEMC protocol as the case-study protocol, and a proposal for a set of guidelines and techniques that one can use to model and verify security properties of PESPs. Although we only use one case-study protocol to show the application of CPN for the purpose of modeling and verification of PESP, the application of CPN as detailed in this paper and
the resulting guidelines are nevertheless the result of experimenting the use of CPN with several other PESPs [17, 18].

The rest of the paper is structured as follows: section 2 discusses related work, section 3 provides some background information of CPN and the PIEMC protocol being modeled and verified in this paper. Section 4 and 6 details the modeling and verification approach for the case-study PIEMC protocol, and Section 7 provides the conclusion and future work.

2 Related Work

The use of formal methods to evaluate security protocols is well documented. In this work we propose the application of formal methods to the new area of PESPs. Al-Azzoni et al [19] propose the use of CPN to model and verify cryptographic protocols. The main differences between their work and the work proposed in this paper are: (1) their work focuses on the cryptographic protocol itself while our work focuses on protocols which apply cryptographic primitives (including cryptographic protocols), and (2) in comparison to the cryptographic protocol being studied by Al-Azzoni et al [19], our work aims at modeling and verifying PESPs which are of a much larger scale involving significantly more message exchanges, and more cryptographic operations. In addition, in this paper, we propose guidelines for capturing cryptographic primitives, their related operations, and set of other generalized techniques (details in Section 4 and 6) that can be usefully applied to modeling and verifying generic PESPs and other protocols that apply cryptographic primitives.

Other related work include the Automated Validation of Internet Security Protocols and Applications (AVISPA) tool [20] and Scyther [21]. These tools are specially built to model and verify security protocols. The main benefit of these tools is that they support the capturing of encryption data, encryption keys, and also include built-in threat models that allow automatic verification of some standard security properties, such as message confidentiality, integrity, and authenticity properties. The problem with these tools is that they work on modeling and verifying network-level threats and security properties. Such limitation is evident from the types of attack models supported and the security properties that can be verified. Furthermore, it is not evident how complex cryptographic primitives (such as group encryption) and their related behaviours can be represented using these tools.

For example, Scyther has built-in threat model which is based on the Dolev and Yao network intruder model [10]. This threat model does not take into account malicious insider attacks, nor application-specific attacks, such as, in the case of the PIEMC protocols, various message or encryption message manipulations that a malicious entity could launch (details in Section 4.2). Consequently, this limits the range of security properties that can be verified to those few standard security properties such as confidentiality, integrity, authenticity, and non-repudiation. As a result, it is difficult to verify application-specific properties; for example, in PESPs, the security properties of interest (as explained in
Section 3) are behavioural by nature and application-specific (not portable across different applications). It is therefore not conceivable how these properties could be automatically verified using the mentioned tools. This limitation is evident from the list of security protocols that have been modeled using AVISPA\textsuperscript{1} or Scyther\textsuperscript{2} whereby the majority of them are authentication protocols. When protocols related to privacy (such as Geopriv [22]) are modeled, the privacy property is normally reduced to confidentiality and authenticity properties. We argue that this is a simplistic approach to verifying privacy properties and that privacy does not simply equate to confidentiality and/or authenticity.

The main reason for the inadequacy of these tools can be attributed to the fact that these tools attempt to, with good and legitimate reasons, abstract as much as possible the security protocol modeling and verification process. However, this abstraction comes at a cost of restrictions and inflexibility in the range of their applications. To cope with PESPs whereby attacks and security properties vary from one application to another, it is almost inevitable for us to go down to a lower-level approach whereby fewer abstractions or automations are available, but more flexibility and wider application range are afforded. Simple Homomorphism Verification Tool (SHVT) [23] is an example of such a tool, which is based on the model-checker approach. SHVT allows one to model a protocol, formalize security goals and attack models, followed by verification of security goals by specifying some transition patterns and checks if they are achievable (see Du [24, Chapter 5 and 6] for applications of the SHVT). There are many similarities between SHVT and the CPN approach this paper; however, we argue that SHVT lacks the graphical interface support that CPN has which makes the benefits associated with having a graphical interface as explained in Section 1.1) unattainable. Besides, SHVT lacks the performance analysis support that CPN has - which makes the use of a SHVT less desirable choice for the purpose of PESPs.

Finally, another type of formal method approach known as process algebra has also been proposed to model and verify security protocols [25, 26]. Readers who are interested in the comparison between process algebra and petri nets should consult the paper by Aalst [27]. In summary, each approach has its own merits; nevertheless, in this paper, we choose the CPN approach because it is relatively simple and easy to understand (as opposed to process algebra where simple things could become very complicated [27]). We do not claim that CPN is better, rather, we prefer to use the CPN approach over process algebra.

3 PIEMC Preliminary

In this section, we introduce a PESP: the Private Information Escrow Bound to Multiple Conditions (PIEMC) protocol. The PIEMC protocol [11] is used in a federated single-sign on (FSSO) environment whereby a user only has to authenticate once to an identity provider (IdP) in order to access services from

\textsuperscript{1} http://avispa-project.org/library/index.html
\textsuperscript{2} http://www.lsv.ens-cachan.fr/Software/spore/
multiple service provider (SP)s. There are 5 main entities involved in the PIEMC protocol: user, IdP, SP, anonymity revocation manager (ARM), and referees. An IdP vouches to the SPs that although the user is anonymous, when certain conditions are fulfilled, the user’s identity can be revealed. Obviously, the nature of the services that SPs provide is such that although a user’s identity information is not required in service delivery, it may be conditionally revealed in the future. An ARM or a group of referees is responsible for revoking the user’s anonymity when conditions are satisfied.

There are two variants of the PIEMC protocol: the first variant (denoted as PIEMC-T) uses a trusted ARM for anonymity revocation (Figure 2). The second variant (denoted as PIEMC-NT) does not use an ARM, instead, a group of referees is used for revoking users anonymity (Figure 3). In this paper, we focus on the the second variant of the protocol. To aid an understanding of the PIMEC-NT model detailed in Section 4, we provide the details of PIEMC-NT in Section 3.1. As PIEMC-T is not the focus of this paper, we only summarize its operation in Section 3.1.

For PIEMC security properties verification purpose detailed in Section 6, a summary of the relevant key security properties is provided in Section 3.3.

3.1 PIEMC-NT

In this section, we describe PIEMC-NT. It consists of four stages: the personally identifiable information (PII) escrow (PE) stage, key escrow (KE) stage, multiple conditions (MC) stage, and the Revocation stage.
PE stage The first stage is the PII escrow stage which is started when a user needs to access some services from a SP - say SP1. Upon receiving the service request from the user, the user and SP1 agree on a set of conditions under which the user’s PII can be revealed - we denote such conditions as Cond1. Next, the PII escrow stage starts:

1. **NT-PE-1**: SP1 sends a request to the IdP for the user’s PII to be escrowed.\(^3\) Included in this request are the agreed conditions Cond1 between the user and SP1 under which the user’s PII can be disclosed.

2. **NT-PE-2**: The user encrypts his/her PII using a Verifiable Encryption (VE) scheme under a freshly generated key pair. The user sends to the IdP the ciphertext (which should contain the PII) and the public key used to produce the ciphertext. At this stage, the user keeps the corresponding private key (that is needed to recover the PII from the ciphertext) undisclosed.

3. **NT-PE-3**: From the given encryption and other data, the user and IdP engage in a cryptographic ‘proof-of-knowledge’ protocol (PK). This PK is used to prove to the IdP that the encryption that the user gives correctly encrypts the user’s PII without letting the IdP learn the value of the PII itself. We denote such an operation a PK-VE operation. The output of a PK-VE is either the acceptance or rejection from the IdP regarding the correctness of the given VE ciphertext.

Note that the proposed modeling and verification approach only model cryptographic primitives behaviours necessary for the PIEMC protocol - we are not interested in the security of the cryptographic primitives themselves.

KE stage Assuming that the PK-VE is successful, the PIEMC protocol continues to the KE stage:

- **NT-KE-1**: the IdP and the user engage in a type of cryptographic PK protocol, called the Direct Anonymous Attestation (DAA) which is used to testify that the user is using a valid TPM device without learning the identity of the TPM device - thus privacy-preserving.

- **NT-KE-2**: the user’s TPM generates a universal custodian-hiding verifiable group encryption (UCHVE) of the VE private key (generated from the previous PE stage) under Cond1 and the corresponding TPM proof. For n referees, such a UCHVE produces n ciphertext pieces to be given to n referees. Out of these n referees, t referees are the designated referees and n − t are the non-designated referees. Only designated referees can decrypt the ciphertext pieces. At least k (k ≤ t) decrypted ciphertext pieces are required in order to recover the VE private key. k is known as the threshold value. These n ciphertext pieces are sent to the IdP.

- **NT-KE-3**: The IdP verifies the proof, and if correct, prepares a response to SP2, which include the VE of PII (from the PE stage) and the UCHVE of the VE private key.

---

\(^3\) An escrow process normally involves encrypting some sensitive information that only a trusted party can decrypt.
SP1 now has the ciphertext of the PII (from the PE stage) and the ciphertext of the corresponding private key (from the KE stage). This allows SP1 to recover the user’s PII when Cond1 is fulfilled with the help of a set of referees (see the Revocation stage in Section 3.1) - SP1 cannot decrypt these ciphertexts at this point.

**MC stage** Next, the protocol assumes that the user goes to another SP2, who, similar to SP1, needs the IdP to escrow the user’s PII such that the user’s PII can be revealed when some agreed conditions Cond2 (Cond1 $\neq$ Cond2). This triggers the start of the MC stage:

- **NT-MC-1**: SP2 requests the IdP to escrow K_PRIV,VE
- **NT-MC-2**: the IdP requests the user’s TPM to produce a new UCHVE ciphertext of the VE private key, this time under Cond2, and the corresponding TPM proof.
- **NT-MC-3**: The user replies with the requested encryption and proof to the IdP.
- **NT-MC-4**: The IdP verifies the proof, and if correct, prepares a response to SP2, which include VE of PII (from the PE stage) and the UCHVE of the VE private key (this time encrypted under Cond2).

SP2 now has the ciphertext of the PII (from the PE stage) and the ciphertext of the corresponding private key (from the KE stage). This allows SP2 to recover the user’s PII when Cond1 is fulfilled with the help of a set of referees (see the Revocation stage in Section 3.1). SP2 cannot decrypt these ciphertexts at this point.

In the same session, the user can go to another service provider, say SP3. In this case, due to the single sign-on property, only the MC stage that has to be executed (the PE and KE stage do not have to be executed again). Since an additional MC stage does not reveal any additional capabilities or properties of the PIEMC protocol, the modeling of additional MC stage operation with SP3 is unnecessary.

We define a **session** as the execution of the PE, KE, and one or more MC stages. A user can execute multiple sessions, that is, those three stages are repeatable as necessary. The number of sessions to be executed by the model is parameterized by the **session** parameter (see Table 2 - last line).

**Revocation stage** A distinct stage in the PIEMC protocol is the revocation stage. This stage is executed as needed, however, at a minimum, we require that the user has at least completed one session. The revocation stage is triggered only when some conditions, say Cond1, are fulfilled.

- **NT-REV-1**: SP1 sends $n$ ciphertext pieces to $n$ referees with Cond1
- **NT-REV-2**: Each referee verifies if Cond1 is fulfilled. If so, try to decrypt the given ciphertext piece. Only designated referees can decrypt the ciphertext piece. If decryption is successful, each designated referee sends the decrypted
ciphertext piece to SP1. When SP1 receives \( k \) or more decrypted pieces, it can recover the VE private key, and subsequently decrypt the VE ciphertext to recover the user’s PII.

### 3.2 PIEMC with trusted ARM

In this section, we summarize the first variant of the PIEMC-T protocol. The main difference between this variant with the one described in Section 3.2 is that in PIEMC-NT without trusted ARM, we (1) replace the referees with a trusted ARM, and (2) we use the identity-based re-encryption scheme instead of the group encryption scheme (denoted as CIPHER\_UCHVE\_KVE) to encrypt \( K_{PRIV\_VE} \) during the KE stage.

**PE stage**

- **T-PE-1**: similar to NT-PE-1 except that in addition to Cond1, SP1 needs to provide some UCHVE encryption parameters to the IdP which the user and SP1 have agreed beforehand. These parameters include three variables: \( n, t, \) and \( k \) (explained in detail in the next section).
- **T-PE-2**: same as NT-PE-2
- **T-PE-3**: same as NT-PE-3

**KE stage**

- **T-KE-1**: the IdP requests the trusted ARM to escrow the VE private key (generated from the PE stage previously).
- **T-KE-2**: similar to NT-KE-1 except that this time the DAA is executed between the user and the ARM.
- **T-KE-3**: the TPM device generates the encryption of the VE private key (generated from the PE stage) using an identity-based proxy-reencryption scheme (IBEPRE) and a proof that act as an evidence that the IBEPRE ciphertext correctly encrypts VE private key under the conditions Cond1 (this refers to the provable execution feature of TPM). This IBEPRE ciphertext cryptographically binds the encryption of the VE private key with Cond1. These data are sent to the ARM.
- **T-KE-4**: the ARM verifies the proof and if the proof is correct, sends the IBEPRE ciphertext of the VE private key to the IdP.
- **T-KE-5**: the IdP prepares a response to SP1. Included in the response message are VE ciphertext of the PII (from the PE stage) and the ciphertext of the corresponding private key.

SP1 now has the ciphertext of the PII (from the PE stage) and the ciphertext of the corresponding private key (from the KE stage). This allows SP1 to recover the user’s PII when Cond1 is fulfilled with the help of the trusted ARM (see the Revocation stage in Section 3.2) - SP1 cannot decrypt these ciphertexts at this point.
MC stage
- T-MC-1: SP2 requests the IdP to escrow $K_{PRIV\_VE}$
- T-MC-2: The IdP contacts the ARM to produce a re-encryption of $\text{CIPHER\_IBEPRE\_KVE}$ which was originally bound to $Cond_1$ such that the new ciphertext is cryptographically bound to $Cond_2$. Note that during a re-encryption process, the ARM does not learn the value of $K_{PRIV\_VE}$.
- T-MC-3: The ARM verifies if such a re-encryption request is valid
- T-MC-4: The user replies with the validity of such a re-encryption request (a 'yes' or 'no' answer)
- T-MC-5: The ARM produces the re-encryption of $\text{CIPHER\_IBEPRE\_KVE}$, this time cryptographically bound to $Cond_2$
- T-MC-6: The IdP the IdP prepares a response to SP2. Included in the response message are $\text{CIPHER\_IBEPRE\_KVE}$ (cryptographically bound to $Cond_2$ and $\text{CIPHER\_VE\_PII}$.

SP2 now has the ciphertext of the PII (from the PE stage) and the ciphertext of the corresponding private key (from the KE stage). This allows SP2 to recover the user’s PII when $Cond_1$ is fulfilled with the help of the trusted ARM (see the Revocation stage in Section 3.2) - SP2 cannot decrypt these ciphertexts at this point.

In the same session, the user can go to another service provider, say SP3. In this case, due to the single sign-on property, only the MC stage that has to be executed (the PE and KE stage do not have to be executed again). Since additional MC stage does not reveal any additional capabilities or properties of a PIEMC protocol, the modeling of additional MC stage operation with SP3 is unnecessary.

Revocation
- T-REV-1: SP1 sends $\text{CIPHER\_IBEPRE\_KVE}$ and $Cond_1$ to the ARM.
- T-REV-2: The ARM verifies if $Cond_1$ is fulfilled, and if so, decrypt $\text{CIPHER\_IBEPRE\_KVE}$ to recover $K_{PRIV\_VE}$, and send it to SP1 who can then use it to decrypt $\text{CIPHER\_VE\_PII}$ to recover the user’s PII.

3.3 Main PIEMC Properties
There are several security properties that we need to verify. These security properties are the same for both variants of the PIEMC protocol. These security properties are:

Multiple Conditions: at the end of every session, a user’s escrowed PII must be bound to multiple sets of alternative conditions (each to be used with different SPs).

Zero-knowledge property: this property has two dimensions: (1) prior to conditions fulfillment, IdP, SP, and referees must not learn the value of the user’s PII but at the same time can be convinced that its encryption is correct; (2) when conditions are fulfilled, the revealed PII must be the same as the one certified in the user certificate.
Enforceable Conditions: a user’s PII should never be revealed before all designated referees (the UCHVE threshold value) agree that some conditions which are cryptographically bounded to the PII are satisfied.

Conditions Abused Resistant: an SP, IdP, or user should not be able to trick the user or ARM to encrypt the user PII, or the VE private key, under a set of conditions different from those agreed between a user and an SP. Similarly, an SP or IdP must not be able to successfully revoke the user’s PII using a set of conditions other than those originally agreed.

Session Unlinkability: SPs and IdP must not be able to link, from the available session artifacts, a user from one session to another.

4 Modeling the PIEMC Protocol

A background of CPN is provided before the model is described in details.

4.1 CPN Preliminary

A CPN consists of two types of nodes, places (drawn as ellipses) and transitions (rectangles), and directed edges known as arcs. A place is typed by a color set and contains collections (multi-sets) of data items called tokens of the same type as the place. A transition represents an event and may have a guard associated with it. The guard is a boolean expression enclosed in square brackets. Arcs connect places to transitions and transitions to places, and are inscribed by expressions comprising variables, constants and functions. Variables are typed and can be assigned values known as binding.

A transition’s input places have arcs going to the transition, while its output places have arcs coming from the transition. A transition is enabled if: 1) sufficient tokens exist in each input place to match each respective input arc inscription when evaluated for a particular binding of its variables, and 2) the transition guard evaluates to true for the same binding. If a transition is enabled, it can occur (or be fired). The occurrence of a transition removes tokens specified by the respective arc inscriptions from input places, and deposits tokens specified by inscriptions on the output arcs into output places. The state of a CPN is called a marking. It consists of tokens distributed on each place of the CPN. Occurrence of transitions represent stage changes.

The hierarchical features of CPNs facilitate the constructions of large models by using a number of CPN modules called pages. Each page is linked to a substitution transition (sub-transition) at a higher level of the model. By means of the hierarchical structuring mechanism it is possible to capture different abstraction levels of the modeled system in the same CPN model.

4.2 Model Description

We have modeled both variants of the PIEMC protocol (see Section 3.2 and 3.1). For each variant, there is one main protocol page, with 4-subpages, each representing the four main stages of the protocol: the PE, KE, MC, and revocation.
stage. Each PE page in both variants has one helper sub-page. For each model, we parameterize the number of repeatable session (including the PE, KE, and MC stage) to be executed. Similarly, the attack model for each variant of the protocol is also parameterized.

<table>
<thead>
<tr>
<th>Cryptographic Data</th>
<th>CPN Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>VE ciphertext of PII</td>
<td>CIPHER_VE_PII</td>
</tr>
<tr>
<td>VE Public / Private key</td>
<td>K_PUB_VE/K_PRIV_VE</td>
</tr>
<tr>
<td>UCHVE ciphertext of K_PRIV_VE (representing all the n ciphertext pieces as a whole)</td>
<td>CIPHER_UCHVE_KVE</td>
</tr>
<tr>
<td>UCHVE ciphertext of K_PRIV_VE (representing an individual K_PRIV_VE ciphertext piece).</td>
<td>CIPHER_UCHVE_KVE_PIECE</td>
</tr>
<tr>
<td>IBEPRE of K_PRIV_VE</td>
<td>CIPHER_IBEPRE_KVE</td>
</tr>
</tbody>
</table>

Table 1: Mapping between cryptographic data to CPN colours

Due to space limitation, an exhaustive description of both models is not possible; instead, selective CPN pages which capture our key modeling approach to PESPs is described. The main page and the PE page for both variants are similar, therefore, we will only describe them once. The KE, MC, and the revocation pages are different between both variants. We choose to describe the KE page and the revocation page for the second variant (without trusted ARM) because they capture the TPM-related operations (the KE page) and because it enables use to show the richer revocation stage operation involving concurrent operations and complex attack scenarios. While we have modeled the rest of the pages (the KE and revocation page for the first protocol variant and the MC pages for both variants) for the purpose of security properties verification, they do not show any additional modeling approach that has not already been captured by the other pages described. Several main CPN colour definitions have been provided in Table 2.

**Main page** Figure 4 shows the main page for both variants of the PIEMC protocol. As described earlier, the protocol starts with a user and a service provider SP1 agreeing on a set of conditions (represented by the transition \( \text{U SP1 GENERATE CONDITIONS} \)) then proceed to execute the PE stage (captured in the subtransition \( \text{PII Escrow} \)), followed by the KE stage (represented by another subtransition \( \text{Key Escrow} \)). Upon completion of the KE stage, the user goes to another service provider SP2 and they both then agree on another set of conditions (represented by the transition \( \text{U SP2 GENERATE CONDITIONS} \)) before starting the MC stage (represented by a subtransition \( \text{Multiple Conditions} \)).

The completion of the MC stage marks the completion of one session. At the end of a session, we store the session data accumulated by all entities. How many session that the user executes is parameterized by the value of \( \text{session} \). As long as the value of the variable \( \text{counter} \) is less than or equal to \( \text{session} \) (note the
colset SP_REQ = record genCond:STRING * conditions1:STRING *
  <other fields omitted for simplicity>
colset SP_REQ_SIG = record message:SP_REQ * key:K_SIGN_GEN;
colset SIGNED_SP_REQ = record message:SP_REQ * signat:SP_REQ_SIG;
colset K_PUB_VE = INT;
colset K_PRIV_VE = INT;
colset K_SIGN_GEN = INT;
colset PII = STRING;
colset LABEL = STRING;
colset COMMITMENT_PII = record message:PII * random:RANDOM;
colset SIGNATURE_GEN = record message:MSG * key:K_SIGN_GEN *
  provable: PROVABILITY;
colset SIGNED_MSG = record message:MSG * signat:SIGNATURE_GEN;
colset CIPHER_VE_PII = record message:PII * key:K_PUB_VE * label:LABEL *
  provable:PROVABILITY;
colset CIPHER_UCHVE_KVE = record message:K_PRIV_VE *
  groupKeys:K_PUB_UCHVE_LIST* desigMembers:DESIG_MEMBERS_LIST *
  k:THRESHOLD * n:INT * t:INT * label:LABEL * provable:BOOL;
colset CIPHER_UCHVE_KVE_PIECE = record message:K_PRIV_VE *
  key:K_PUB_UCHVE * label:LABEL * isDesignated:BOOL;
colset SP_RESPONSE = record pseudo:STRING * cipherVE:CIPHER_VE_PII *
  cipherUCHVE:CIPHER_UCHVE_KVE;
colset SP_RESPONSE_SIG = record message:SP_RESPONSE * key:K_SIGN_GEN *
  provable:BOOL;
colset SIGNED_SP_RESPONSE = record message:SP_RESPONSE *
  signat:SP_RESPONSE_SIG;
colset DEC_REQ = record conditions:LABEL *
  uchvePiece:CIPHER_UCHVE_KVE_PIECE;
colset DEC_REQ_SIGNATURE = record message:DEC_REQ *
  key:K_SIGN_GEN * provable:BOOL;
colset SIGNED_DEC_REQ = record message:DEC_REQ * signat:DEC_REQ_SIGNATURE;

Parameters:
val USER_ATTACK1 = false;
val USER_ATTACK2 = false;
val USER_ATTACK3 = false;
val USER_ATTACK4 = false;
val SP_ATTACK1 = false;
val SP_ATTACK11 = false;
val SP_ATTACK12 = false;
val SP_ATTACK2 = false;
val SP_ATTACK22 = false;
val SP_ATTACK3 = false;
val SP_ATTACK4 = false;
val SP_ATTACK5 = false;
val SP_ATTACK6 = false;
val IDP_ATTACK4 = false;
val REF_ATTACK1 = false;
val REF_ATTACK2 = false;
val condActually = true;
val fulfilled =
  if REF_ATTACK1 then true else
  if SP_ATTACK4 then false else
  condActually;
val session=2;

Table 2: Colour Definition for PIEMC
Fig. 4: Main PIEMC page for both variants
guard function of the transition U SP1 GENERATE CONDITIONS), the model will execute another session. Otherwise, the guard function will disable the transition, instead, a token will be placed at the place SP1 REVOCATION CONDITIONS FULFILLED (note the arc inscription from the transition STORE SESSION DATA to the mentioned place) which triggers the start of a revocation stage. The end result of the revocation stage is the success or failure of the revelation of the user’s PII as represented by the existence (or non-existence) token in the place RECOVERED USER PII.

![DIagram](image)

**Fig. 5: PE page for both variants of PIEMC**

**PE page** This page represents the PE stage for both variants of the PIEMC protocol (see Figure 4.2). The message T-PE-1 is represented by the place SP1 PII REQ SIGNATURE, which is of type SIGNED_SP_REQ. From Table 2, this message represents a cryptographically signed message by SP1 and the main content of the message is the conditions Cond1 under which the user’s PII can be revealed. There are other values included in this message, however, they are omitted for simplicity.
Next, the signed message is sent to the IdP who first verify the signature valid. If it is valid, the user sends the T-PE-2 message (represented by the transition U SENDS PII ESCROW DATA). There are three data being sent in this message: the commitment (a type of cryptographic data) of the user PII (represented by the place IDP RECV PII COMMIT), the user-generated one-time VE public key (represented by the place IDP RECV ONE TIME VE PUB KEY), and the ciphertext of the user’s PII (represented by the place IDP RECV PII VE CIPHER).

Following this, the user and the IdP both engage in the PK-VE operation (represented by the transition IDP_U PK PII VE CIPHER). This transition captures the T-PE-3 messages. The output of this transition is a boolean value representing the result of executing the PK-VE operation: if it is true, then the IdP is convinced that the given ciphertext correctly encrypts the user’s PII under the given conditions and public key.

**Key Modeling Techniques**
In this page, we propose several techniques to model the commonly used cryptographic operations in not only PESPs protocols, but also other protocol that apply cryptographic primitives: message signature and verification, message encryption, zero-knowledge proof protocol, and generation of one-time data.

Message signature and verification is commonly used by a message recipient to verify that the message received has not been modified in transit (message integrity), and to verify that the message is indeed sent by the claimed mes-
sage sender (message authenticity). Line 1-4 of Table 2 show an example of how we could build the colour definition for signed messages. First, we define the colour for the message to be signed (line 1). Next, we define the signature of the message. A signature is represented as a tuplet consisting of the message itself, and the signature key (line 2-3). To anticipate the use of more complex signature algorithm, we also include the provability property as part of the signature tuplet. Having defined the message and the signature, we can now define the colour of a signed message, which is simply a tuplet consisting of the message and its signature (line 4).

Having defined the colour representing a signed message, we now need to express its corresponding operations: message signing and verification. These operations are encapsulated in two functions as shown in Table 3. The function \texttt{sign} takes two parameters: a message to sign, and a signature key. The output is a record representing a message signature. The function \texttt{verify} takes two parameters: a signed message, and a signature verification key. To capture the verification of the integrity of a message, it is sufficient to compare the plaintext message in the signed message with the actual message being signed (encapsulated within the message signature). To capture the verification of the authenticity of the message, it is sufficient to compare the key used to sign the message (again, encapsulated within the message signature) with the signature verification key corresponding to the claimed message source. If both of these comparisons succeed, then we have verified the integrity and authenticity of the message. The \texttt{verify} function can be used as a guard to a transition to capture the situation whereby a protocol halts prematurely if a signature verification fail. This is precisely how it is being used as depicted in Figure 4.2 (note the guard function for transition \texttt{STORES PSEUDONYM}).

\begin{verbatim}
fun sign(msg:MSG, signKey:K_SIGN_GEN) =
  (message=msg, key=signKey, provable=false);

fun verify(sigMSG:SIGNED_MSG, verifyKey:K_VERIFY_GEN)=
  #message(sigMSG) = #message(#signat(sigMSG)) andalso
  #key(#signat(sigMSG)) = verifyKey;
\end{verbatim}

Table 3: Capturing Message Signature and Verification

To represent a VE ciphertext, we have defined an appropriate colour (line 15-16 of Table 2). A VE encryption is a record consisting of four fields: the message itself, the public encryption key, the label under which the message is

\footnote{Normally, in order to verify a signature, a message recipient needs both plaintext value of the message and the signature. When a signature has the provable property, a message recipient does not learn the value of the message nor the signature; however, the message recipient can still verify that the message sender knows some valid messages that have been signed by a known entity.}

encrypted, and the provability property. A provable ciphertext means that the recipient of the message can be convinced that the received ciphertext correctly encrypts some claimed value (in this case the user’s PII) without the recipient learning the value of the PII itself nor of the decryption key - also known as zero-knowledge proof (PK). We treat the message field inside a record representing a ciphertext to be unreadable.

The operations related to VE ciphertext include the message encryption and decryption, and PK-VE. The encryption and decryption operations are encapsulated in two function called `veEnc` and `decVE` respectively (see Table 4). The encryption function takes three parameters: the message to encrypt, the public encryption key, and the label under which the message is to be encrypted, and it simply returning a record representing a ciphertext as defined by the colour `CIPHER_VE_PII`, setting the provability property to ‘true’ as this encryption algorithm allows its content to be proven in a zero-knowledge manner. The decryption function takes three parameters: the VE ciphertext, the decryption key, and the encryption label. First, it checks if the decryption key is a correct key (by comparing it with the encryption key encapsulated in the ciphertext). Next, it checks if the label used to decrypt is the same as the label used during the encryption time. If so, a decryption will succeed, and thus, the content of the message can be extracted (represented as the output of the function).

Next, to capture the PK property, we encapsulate it in a transition `IDP_U PK PII VE CIPHER`. Note that all of the input places connected to this transition represent the required data at both the user side (right hand side of the transition) and the IdP side (left hand side of the transition). The behaviour of this PK-VE is captured in the output inscription arc from this transition to the place `IDP_U PK PII VE CIPHER`. This arc inscription captures the essentials verification that a valid PK-VE should verify; for example, the arc inscription verifies that the encrypted data inside the ciphertext (which is unreadable to the IdP) contains the same PII as what is inside a valid user’s certificate (line 7 Table 5). The output of such an operation can be used as a guard to a transition which should not be executed if the PK-VE operation fails. In other words, the result is used to halt the execution of a protocol prematurely as depicted by the guard function in the transition `IDP GENERATES STORES PSEUDONYM AND SEND TO USER`.

| veEnc(msg:PII, pubKey:K_PUB_VE, cond:LABEL)= | {message=msg, key=pubKey, label=cond, provable=true}; |
| decVE(key:K_PRIV_VE, cipherVE:CIPHER_VE_PII, cond:LABEL)= | if key=#key(cipherVE) andalso cond = (#label(cipherVE)) then 1'(#message(cipherVE)) else empty; |

Table 4: Capturing Message Encryption and Decryption
if #provable(idpVeCipher) andalso
#provable(cert) andalso
#signKey(cert) = #certVerifyKey(
#verifyKeys(readIDPRecord("idp.txt"))) andalso
#message(idpCommitment) = #pii(cert) andalso
#message(idpVeCipher) = #message(idpCommitment) andalso
#message(idpVeCipher) = #pii(cert) andalso
idpVeKey = #key(idpVeCipher) andalso
idpGenCond = #label(idpVeCipher) then 1'true
else 1'false

Table 5: Capturing PK-VE Behaviours

Finally, in this page, we also propose a technique to ensure that we can generate one-time data in a simple manner and the generated data is guaranteed to be unique. Figure 6 shows the process of generating one-time data that are subsequently used for other operations (such as the generation of a VE ciphertext). Our approach is simply to encapsulate the generation of one-time data in a function with the help of a text file. This text file stores the next valid one-time data. When this data is used, we then update the file with the next valid one-time data (by simply incrementing it by 1), and so on. See Table 5 for the detail of the function. By using this function, at any point in the model, we only have to call the function to obtain a unique one-time data that has never been used before.

fun getRandom() = 
let
  val random_file = TextIO.openIn("random1.txt")
  val count = TextIO.inputLine(random_file)
  val _ = TextIO.closeIn(random_file)
  val update = valOf(Int.fromString(count)) + 1
  val _ = TextIO.output(random_file2, Int.toString(update))
  val _ = TextIO.closeOut(random_file2)
in
  count
end;

Table 6: Get one-time data function

KE Page for PIEMC without trusted ARM Figure 4.2 shows the KE page model. The message NT-KE-1 is represented by the transition TPM GENERATES AIK KEY SIGNED SESSION KEY AND DAA PROOFS. We do not model the details of a DAA protocol as we are only concerned with the output of such a protocol: convincing the IdP that the user is using a valid TPM module without learning the permanent identity of the TPM device itself - see Section 3.2. Instead, a two sets of signing and verification keys representing the TPM device are produced:
a set of attestation identity key (AIK) keys (used to sign messages generated from within the TPM device), and a set of session keys (to sign other messages). These two sets of keys are used throughout the PIEMC protocol.

After establishing the use of correct TPM device, the user’s TPM starts the execution of Module2 to produce a UCHVE of K_PRIV_VE (represented by the transition TPM GENERATES Cipher UCHVE KVE). After the execution of Module2, the user’s TPM device generates a proof of correct execution (represented by the transition TPM GENERATES CORRECT EXECUTION PROOF) which is signed using the TPM AIK key. Message Nt-KE-2 is represented by the sending of the UCHVE of K_PRIV_VE and the corresponding TPM proof to the IdP (note the two outgoing arcs from the transition U SENDS ARM MODULE 1 RESULT AND TPM PROOF).

Upon receiving the UCHVE of K_PRIV_VE and the proof, the IdP verifies the given encryption and the proof (represented by the transition IDP VERIFIES MODULE 2 TPM PROOF). If it succeeds, it then prepares a response message to the SP1 (represented by the transition IDP STORES AND PREPARES SP RESPONSE). Message Nt-KE-3 is represented by a colour SIGNED_SP_RESPONSE (see Table 2).

Key Modeling Technique. In this page, the key modeling technique proposed is the representation of the TPM module execution with provable execution property [16]. What is of interest is how we could represent the generation and verification of TPM correct module execution as this is one of the crucial operations on which the security properties of our protocol depend.
In general, the proof of a correct module execution is represented by three factors: a TPM-signed representation of the input, output, and the module code-base that reside in the protected TPM registry. Therefore, in the generation of the proof, we use a multi-step approach. The registry value for Module2 code and its input are represented in the variable \( tpmProof \) which is form as follows (note the output arc from the transition TPM EXECUTES MODULE 2 to the place TPM PROOF DATA:

- the Module2 registry value is represented as a simple "Module2PCRValue" string,
- for each of the input to Module2, we obtain its string representation (using the built-in COLOUR.mkstr function), then
- we concatenate all of the above string to obtain the first part of the proof.

The next piece of the proof requires the output of Module2, therefore, we need to execute Module2 first before we can form the complete proof. After executing Module2 (represented by the transition TPM EXECUTES MODULE2 and its corresponding code region shown in Table 7 top part), we obtain the output: the UCHVE of K PRIV VE. We then proceed to form the complete TPM proof by concatenating \( tpmProof \) with the String representation of the UCHVE of K PRIV VE. The complete TPM proof is then signed using the TPM AIK signing key. The operation of forming and signing the TPM proof is performed by the code region linked to the transition TPM GENERATES CORRECT EXECUTION PROOF - see Table 7 - bottom part.

The signed TPM proof as well as the output from Module2 are then sent to the IdP as represented by the transition U SENDS IDP MODULE 2 RESULT AND TPM PROOF.

Upon receiving the UCHVE of K PRIV VE and the corresponding TPM proof, the IdP then proceeds to verify if the proof is correct. To do so, the IdP has to first verify if the correctness of the signed TPM proof. If so, it then needs to verify that the TPM proof is similar to the one that it generates independently. Based on the details provided in [16], a verifier can generate a valid TPM proof as follows: the Module2 code base and input to Module2 are known to the IdP. Therefore, to represent the generation of a valid TPM proof, the IdP only has to concatenate the Module2 code base registry value (represented as a string "Module2PCRValue") with the input values that it knows and the received string representation of UCHVE of K PRIV VE. If the generated TPM proof is the same as the one received, then the IdP is convinced that the received UCHVE of K PRIV VE is valid.

MC page For completion, the MC page for the PIEMC protocol without trusted ARM is provided in Figure 8.

Revocation page for PIEMC without trusted ARM Figure 9 shows the model for the Revocation stage. The revocation stage starts when one of the
Code region for transition TPM EXECUTES MODULE 2:

```plaintext
input ();
output (uchveCipher1);
action
let
val userRec = readUserRecord("user.txt")
val groupKeys = if not USER_ATTACK3 then #uchvePubKeys(userRec) else [7,8,9]
val msg = if not USER_ATTACK3 then #priv(#veKeys(userRec)) else 0
val desigMembers = if not USER_ATTACK3 then #desigMembers(userRec) else [7,8,9]
val k = if not USER_ATTACK3 then #k(userRec) else 99
val t = if not USER_ATTACK3 then length (#desigMembers(userRec)) else 99
val label = if not USER_ATTACK4 then #conditions1(#conditions(userRec)) else "hardToFulfillConditions"
val pub = #pub(#veKeys(userRec))
val correct = msg = pub
val uchveCipher1 = if not (USER_ATTACK4 orelse USER_ATTACK3) then uchveEnc(msg, groupKeys, correct, desigMembers, t, k, n, label) else {message=msg, groupKeys=groupKeys, desigMembers=desigMembers, t=t,k=k, n=n,label=label,provable=true} in uchveCipher1 end;
```

Code region for transition TPM GENERATES CORRECT EXECUTION PROOF:

```plaintext
input (uchveCipher1, tpmProof);
output (signedTPMProof);
action
let
val userRec = readUserRecord("user.txt")
val aikKey = #privAIK(#tpmKeys(userRec))
val completeTPMProof = if not USER_ATTACK3 andalso not USER_ATTACK4 then CIPHER_UCHVE_KVE.mkstr(uchveCipher1)^tpmProof else CIPHER_UCHVE_KVE.mkstr(uchveCipher1)^"arbitraryTPMProof" val sigTPMProof = sign(completeTPMProof, aikKey) val signedTPMProof = {message=completeTPMProof, signat = sigTPMProof} in signedTPMProof end;
```

Table 7: Module2 Execution and Generating TPM proof
Fig. 8: MC page - PIEMC without trusted ARM

Fig. 9: Revocation Page for PIEMC without trusted ARM
service providers (in this case SP1) discovers that some conditions are satisfied. The message NT-REV-1 is represented as a colour of type SIGNED_DEC_REQ and is sent from SP1 to all referees involved (transition SP1 SIGNS FULFILLED CONDITIONS). We model three referees in this page.

Upon receiving the message, each referee verifies if the condition string sent by SP1 earlier has been satisfied. Whether the conditions are fulfilled or not is parameterized in two parameters: fulfilled and condActually (see Table 2). The differences between these two parameters relate to the assumed attack behaviours from the referees which are explained later. When each of the referees believe that the conditions are fulfilled, it then proceeds to decrypt the given ciphertext pieces, and send the decrypted pieces to SP1 (this corresponds to message NT-REV-2 in Figure 3). If SP1 receives $t$ or more decrypted pieces, it can then recover $K_{PRIV,VE}$ and proceed to decrypt CIPHER_VE_PII to obtain the user’s PII.

**Key Modeling Technique** The main modeling technique introduced in this page is the representation of operations involved in the decryption of threshold encryption (such as the UCHVE). As stated earlier in Section 3.1, we represent group encryption ciphertext in two forms: CIPHER_UCHVE,KVE and CIPHER_UCHVE,KVE,PIECE. Dealing with the individual pieces of a group ciphertext could complicate the model, and therefore, we have only been representing such an encryption in its group form. However, during the decryption process, we need to represent such an encryption in its individual pieces, each to be sent to its corresponding referee. Therefore, we introduce a conversion function that converts the group representation of such a ciphertext to its corresponding individual pieces - as shown in Table 8.

Next, we provide a place to store all of the decrypted ciphertext pieces. At this point, we are only concerned on the quantity of decrypted pieces available, and thus, the decrypted ciphertext can be sufficiently represented by a simple primitive, such as a boolean value with ‘true’ representing successful decryption, and ‘false’ representing failed decryption (as described in Section 3.1, only designated referees can successfully decrypt a ciphertext piece).

After receiving the decryption response from each of the referees (success or failure), the SP1 needs to filter out those successful decrypted pieces from those failed ones (represented by the transition SP1 FILTERS DECRIPT RESULT). After isolating the successful decryption pieces, it then checks if it has at least $t$ number of such pieces. If so, it can then proceed to recover the data hidden in the group ciphertext. Such process is captured by the input arc inscription from the place UCHVE PIECE DECRYPT SUCCESS to the transition SP1 RECOVERS VE PRIVATE KEY whereby only when there are $t$ tokens in the mentioned place will the corresponding transition be enabled.

This modeling technique takes advantage of the CPN capability to easily capture the boundedness property of an operation. More precisely, in this scenario, the boundedness property is directly translated to the threshold property which commonly exists in many cryptographic primitives used in PESP or other security protocols in general.
Capturing Session Data Common to many security protocols, not limited to PESP, when there are several message exchanges between various entities, it is important that we keep track of the data being exchanged and store them in such manner that they can be retrieved easily. In modeling the PIEMC protocol, we propose the use of text files as the method to keep track of the session data being exchanged in a session. To allow easy access to these text files, we propose the following approach:

- Decide on the data that needs to be stored by each entity from each session,
- translate those data into one or more CPN colour of type record (preferably one colour type for each entity) (for example, see Table 9 top part),
- create a function that reads a text file into a CPN variable of some record type as defined earlier (see Table 9 - middle part),
- when necessary, the fields of the record (now represented as a CPN variable) should be updated based on some received or generated data (see Table 10 - top part),
- create a function that allows the updated CPN variable to be written back to the relevant text file (see Table 9 - bottom part).
- At the end of each session, dump the session variable (holding the session data) into a new file representing an archive of a session data before starting a new session (see Table 10 bottom part).

Using the above mechanism, one could easily read, store, and update session data throughout the session without having to maintain tokens in various places across multiple CPN pages. This also reduces the need to do the ‘vacuum cleaner’ functionality at the end of each session to remove session tokens from the relevant places. Such ‘vacuum cleaner’ functionality is known to be inefficient in a petri net model [27].

Adding attack model There are numerous types of attacks that one could think of in order to attack a protocol. In our model, we limit ourselves to modeling those attacks that originate from legitimate entities as opposed to those from external entities. Our approach to modeling attacker behaviour is to parameterize them such that we can switch certain attack on or off depending on the configuration. We have modeled about 15 different attacks originating from either the user, SP, ARM, and referees (see Table 2). In this section, we elaborate several attack models that we have included in the PE page, KE page, and the revocation page.

```haskell
fun convertToUCHVEPiece(uchve:CIPHER_UCHVE, refPubKey:K_PUB_UCHVE) =
  {message=(#message(uchve)), key= refPubKey,
   label=(#label(uchve)), isDesignated=List.exists (fn y => refPubKey=y) (#desigMembers(uchve))}
```

Table 8: Converting group ciphertext form to individual ciphertext form


```plaintext

2 fun readUserRecord(file:STRING)=
3   let
4     val ins = TextIO.openIn(file)
5     val userRec = USER_RECORD.input(ins)
6     val _ = TextIO.closeIn(ins)
7   in
8     userRec
9   end;
10
11 fun updateUserRecord(file:STRING, userRec:USER_RECORD) =
12   let
13     val os = TextIO.openOut(file)
14     val _ = USER_RECORD.output (os, userRec)
15     in
16     TextIO.closeOut(os)
17   end;
18
19 fun readSPRecord(filename:STRING) =
20   let
21     val _ = TextIO.openIn(filename)
22     val rec = SP_RECORD.input()
23     val _ = TextIO.closeIn(filename)
24   in
25     rec
26   end;
27
28 fun updateSPRecord(filename:STRING, rec:SP_RECORD) =
29   let
30     val _ = TextIO.openOut(filename)
31     val _ = SP_RECORD.output (os, rec)
32     in
33     TextIO.closeOut(os)
34   end;

Table 9: Functions to read and update session data
```

```plaintext
1 Code region for transition \texttt{SP1 VERIFIES AND STORES RESPONSE} (from page \ac{KE}) : 
2 input (signedSPResponse);
3 output ();
4 action
5 let
6   val spRec = readSPRecord("sp1.txt")
7   val spResponse = #message(signedSPResponse)
8   val pseudo = #pseudo(spResponse)
9   val cipherVE = #cipherVE(spResponse)
10  val cipherUCHVE = #cipherUCHVE(spResponse)
11  val spRec = SP_RECORD.set_pseudo spRec pseudo
12  val spRec = SP_RECORD.set_cipherVE spRec cipherVE
13  val spRec = SP_RECORD.set_cipherUCHVE spRec cipherUCHVE
14  in
15  updateSPRecord("sp1.txt", spRec)
16 end;
17
18 Code region for transition \texttt{STORE SESSION DATA} (from the main \ac{PIEMC} page) : 
19 input (counter);
20 output ();
21 action
22 let
23   val userRec = readUserRecord("user.txt")
24   val idpRec = readIDPRecord("idp.txt")
25   val armRec = readARMRecord("arm.txt")
26   val sp1Rec = readSPRecord("sp1.txt")
27   val sp2Rec = readSPRecord("sp2.txt")
28   val _ = updateUserRecord("user_sess":"Int.toString(counter-1)"".txt", userRec)
29   val _ = updateIDPRecord("idp_sess":"Int.toString(counter-1)"".txt", idpRec)
30   val _ = updateArmRecord("arm_sess":"Int.toString(counter-1)"".txt", armRec)
31   val _ = updateSPRecord("sp1_sess":"Int.toString(counter-1)"".txt", sp1Rec)
32   val _ = updateSPRecord("sp2_sess":"Int.toString(counter-1)"".txt", sp2Rec)
33 end;
34
Table 10: The process of of reading, modifying, updating, and storing session data
```
PE Attack Model In this page, there are several parameterized attacks included. See the list of parameter declaration in Table 2. In this paper, we show two examples of attacks:

– USER ATTACK1: user gives incorrect encrypted PII and conditions during PE stage. This attack is expressed in Figure 6 as arc expression from the transition \( U \) generates PII cipher to the place PII VE cipher.
– USER ATTACK2: user gives incorrect VE public key to the IdP. This attack is expressed in Figure 4.2 as an arc expression from the transition \( U \) sends PII escrow data to the place IdP recv one time VE pub key.

KE Attack Model In relation to this page, there are several parameterized attack modeled:

– USER ATTACK3: a user executes Module2 under incorrect parameters necessary for a correct UCHVE of K_PRIV,VE, and
– USER ATTACK4: a user could execute Module2 and bind the encryption with a set of conditions Cond1 which are different from the originally agreed conditions Cond1 and are difficult or impossible to fulfill.

The above two attacks are expressed in the code region for the transitions TPM executes module 2 and TPM generates correct execution proof - see Table 7.

Revocation Page Attack Model There are several attack behaviours that we have parameterized into the model at the KE page. We assume that Referee 1 and Referee 3 are malicious, while Referee 2 is honest. The value fulfilled refers to the decision that a referee made regarding the (non-)fulfillment of a given condition. The value condActually refers to the actual (non-)fulfillment of some conditions. Several attacks are elaborated here:

– REF ATTACK1: Referees agree that a condition is fulfilled while it is not. This attack is expressed in the declaration of the constant fulfilled (see Table 2): when this attack is enabled, then we set the value of fulfilled to true regardless of the value cond Actually. Thus, this has the effect of malicious referee 1 and 3 to return a ‘true’ value as an output from the transition REF1 verify conditions fulfillment - which subsequently means enable the decryption of the given ciphertext piece. The honest referee 2, on the other hand, always returns the actual (non-)fulfillment of the conditions as set by the parameter condActually.

– REF ATTACK2: Referees collude among themselves to recover K_PRIV.VE. This attack is expressed in the arc expression from the transition REF1 decrypts and sends uchve piece and REF3 decrypts and sends uchve piece to the place REFEREES UCHVE PIECES Decrypt Extra Copy to mimic the action of malicious referees colluding with each other in an attempt to recover K_PRIV.VE.
– SP_ATTACK3: SP1/SP2/IDP attempts to start revocation stage using wrong easy-to-fulfill conditions. This action is reflected in the arc expression from the transition SP1 RETREIVES FULFILLED CONDITIONS to the place CONDITIONS FULFILLED. Of course, when this attack is modeled, we need to set the value of condActually to ‘true’.
– SP_ATTACK4: SP1/SP2/IDP attempts to start revocation stage using correct but not fulfilled conditions. When this attack is modeled, the value of fulfilled and condActually is thus set to ‘false’ (see Table 2).

**Other Attacks** There are other types of attacks which have been parameterized into the model, but are not detailed in this paper:
– SP_ATTACK1: SP1 uses easy-to-fulfill conditions that were not agreed prior with the user during PE stage.
– SP_ATTACK11: SP1 uses easy-to-fulfill conditions that were not agreed prior with the user during KE stage.
– SP_ATTACK12: SP1 uses other non-agreed UCHVE parameters
– SP_ATTACK2: SP2 uses easy-to-fulfill conditions that were not agreed prior with the user.
– SP_ATTACK22: SP2 uses other non-agreed UCHVE parameters
– SP_ATTACK5: SP1 uses invalid signature key to sign SP1 request message during PE stage
– SP_ATTACK6: SP2 uses invalid signature key to sign SP2 request message during MC stage

5 Model Validation

Before we start performing security behaviour verification, we firstly need to perform model validation. There are several basic properties that we need to verify to ensure that the PIEMC model is a faithful representation of the protocol specification. To do so, there are seven basic properties that we need to verify for the PIEMC without trusted ARM. For completeness, each of these properties is detailed, along with the corresponding session data and/or state-space analysis queries.

5.1 Property-1

This property establishes the correctness of users’ behaviours which is as follow: users should generate correct one-time VE key pair, correct conditions upon which user’s PII can be revoked, correct VE encryption of the PII, and correct UCHVE encryption (and their corresponding re-encryptions) of the VE private key.

Specifically, we define ‘correct’ values as follows (for both session 1 and session 2):

– a one-time VE key pair is correct if the values of both the public and private keys are the same (represented as an integer value),
– to represent correct conditions, we require that the value of the condition string to be in the format of: "Conditions SP"<SP Identifier: 1 or 2><random one-time number that is greater than 0>
– to represent a correct encryption of user PII, we require that the value of the ‘message’ field of the user generated VE cipher contains the same value as the one in the user’s certificate, and that the key and label used to encrypt the PII is the same as the ones generated
– to represent correct encryption of VE private key, the message contained in the UCHVE encryption as a result of execution of Module2 must contain the correct VE private key as generated by the user, and that the key for such a UCHVE encryption must be the value of the condition string for SP1 and SP2 (for session 1 and 2 respectively)

This property can be verified using a combination of session-data analysis and state-space analysis as shown in Table 5.1.

5.2 Property-2

Property-2 is about verifying the correctness of the PE stage. Assuming that Property-1 holds, at the end of PE stage (for both session 1 and session 2), a valid model of the protocol has to be in the following states:

– state2-1: the IDP must have a correct VE cipher of the user PII - which is consistent with the one that user generated earlier,
– state2-2: the IDP must have the correct one-time per session VE public key that the user generated,
– state2-3: the IDP must have the correct condition string for SP1 that is consistent with the one agreed between user and SP1,
– state2-4: the PK PII VE operation must not fail (given that we do not any malicious behaviours yet), and
– state2-5: a unique one-time session-pseudonym must be generated by IDP and correctly saved by the user.

This property is verified using a combination of session-data and state-space analysis as shown in Table 12

5.3 Property-3

Property-3 establishes the correctness of the KE stage. Assuming that Property-1 holds, at the end of the KE stage (for both session 1 and session 2) the protocol should be at the following states:

– state3-1: the ARM must have received correct public AIK key, as well as correct public signature verification session key from the user’s TPM
– state3-2: the DAA operation must not fail (given that we do not consider the use of corrupted TPM in which case the DAA operation will fail),
val os = TextIO.openOut("property1.txt")

fun checkConditions(cond:STRING) = 
(substring(cond, 0, 14) = "Conditions SP1" orelse substring(cond, 0, 14) = "Conditions SP2") andalso 
valOf(Int.fromString(substring(cond, 14, (String.size(cond) - 14)))) > 0

val userRec1 = readUserRecord("user_sess1.txt")
val userRec2 = readUserRecord("user_sess2.txt")

val sameKey1 = #pub(#veKeys(userRec1)) = #priv(#veKeys(userRec1))
val sameKey2 = #pub(#veKeys(userRec2)) = #priv(#veKeys(userRec2))

val keyOK = sameKey1 andalso sameKey2
val cond1_1 = #conditions1(#conditions(userRec1))
val cond2_1 = #conditions1(#conditions(userRec2))
val cond1_2 = #conditions2(#conditions(userRec1))
val cond2_2 = #conditions2(#conditions(userRec2))
val condOK11 = checkConditions(cond1_1)
val condOK21 = checkConditions(cond2_1)
val condOK12 = checkConditions(cond1_2)
val condOK22 = checkConditions(cond2_2)
val random1 = valOf(Int.fromString(substring(cond1_1, 14, (String.size(cond1_1) - 14))))
val random2 = valOf(Int.fromString(substring(cond2_1, 14, (String.size(cond2_1) - 14))))
val random3 = valOf(Int.fromString(substring(cond1_2, 14, (String.size(cond1_2) - 14))))
val random4 = valOf(Int.fromString(substring(cond2_2, 14, (String.size(cond2_2) - 14))))
val randomOK = random1 <> random2 andalso random1 <> random3 andalso random1 <> random4
andalso random2 <> random3 andalso random2 <> random4 andalso random3 <> random4
val condOK = condOK11 andalso condOK21 andalso condOK12 andalso condOK22 andalso randomOK
val cert1 = #cert(userRec1)
val cert2 = #cert(userRec2)

val veCipherMarks = SearchNodes(
EntireGraph, fn n => length (Mark.PE'PII_VE_CIPHER 1 n) > 0,
NoLimit, fn n => n, [], op :: );
val veCipher1 = List.nth (Mark.PE'PII_VE_CIPHER 1 8, 0)
val veCipher2 = List.nth (Mark.PE'PII_VE_CIPHER 1 46, 0)
val messageOK = cert1 = cert2 andalso #message(veCipher1) = #pii(cert1)
andalso #message(veCipher2) = #pii(cert2)
val keyOK = #key(veCipher1) = #pub(#veKeys(userRec1)) andalso
#key(veCipher2) = #pub(#veKeys(userRec2)) andalso
val labelOK = #label(veCipher1) = #genCond(#conditions(userRec1)) andalso
#label(veCipher2) = #genCond(#conditions(userRec2))
val veCipherOK = messageOK andalso keyOK andalso labelOK
val uchveEncMarks1 = SearchNodes(
EntireGraph, fn n => length (Mark.KE'TPM_GENERATED_CIPHER_UCHVE_KVE 1 n) > 0,
NoLimit, fn n => n, [], op :: );
val uchve11 = List.nth (Mark.KE'TPM_GENERATED_CIPHER_UCHVE_KVE 1 17, 0)
val uchve12 = List.nth (Mark.KE'TPM_GENERATED_CIPHER_UCHVE_KVE 1 55, 0)
val messageUchveOk1 = #message(uchve11) = #priv(#veKeys(userRec1)) andalso
#message(uchve12) = #priv(#veKeys(userRec2)) andalso
val labelUchveOk1 = #label(uchve11) = cond1_1 andalso #label(uchve12) = cond1_2
val uchveEncMarks2 = SearchNodes(
EntireGraph, fn n => length (Mark.MC'TPM_GENERATED_CIPHER_UCHVE_KVE 1 n) > 0,
NoLimit, fn n => n, [], op :: );
val uchve21 = List.nth (Mark.MC'TPM_GENERATED_CIPHER_UCHVE_KVE 1 17, 0)
val uchve22 = List.nth (Mark.MC'TPM_GENERATED_CIPHER_UCHVE_KVE 1 55, 0)
val messageUchveOk2 = #message(uchve21) = #priv(#veKeys(userRec1)) andalso
#message(uchve22) = #priv(#veKeys(userRec2)) andalso
val labelUchveOk2 = #label(uchve21) = cond2_1 andalso #label(uchve22) = cond2_2
val uchveOK = messageUchveOk1 andalso labelUchveOk1 andalso messageUchveOk2
andalso labelUchveOk2
val property1 = keyOK andalso condOK andalso veCipherOK andalso uchveOK;
if property1 then TextIO.output (os, "Property 1 is SATISFIED\n") else
TextIO.output (os, "Property 1 is NOT SATISFIED\n");
TextIO.closeOut os;

Table 11: Session-data and state-space analysis for Property-1
val os = TextIO.openOut("property2.txt")
val userRec1 = readUserRecord("user_sess1.txt")
val userRec2 = readUserRecord("user_sess2.txt")
val idpRec1 = readIDPRecord("idp_sess1.txt")
val idpRec2 = readIDPRecord("idp_sess2.txt")
val sp1Rec1 = readSPRecord("sp1_sess1.txt")
val sp1Rec2 = readSPRecord("sp1_sess2.txt")
val sp2Rec1 = readSPRecord("sp2_sess1.txt")
val sp2Rec2 = readSPRecord("sp2_sess2.txt")
val cond1_1 = #conditions1(#conditions(userRec1))
val cond2_1 = #conditions2(#conditions(userRec1))
val cond1_2 = #conditions1(#conditions(userRec2))
val cond2_2 = #conditions2(#conditions(userRec2))
val veCipherMarks = SearchNodes(
    EntireGraph, fn n => length (Mark.PE'PII_VE_CIPHER 1 n) > 0, NoLimit, fn n => n, [], op :: );
val veCipher1 = List.nth (Mark.PE'PII_VE_CIPHER 1 8, 0)
val veCipher2 = List.nth (Mark.PE'PII_VE_CIPHER 1 46, 0)
val idpVeCipher1 = #cipherVE(idpRec1)
val idpVeCipher2 = #cipherVE(idpRec2)
val idpVeCipherOK = idpVeCipher1 = veCipher1 andalso idpVeCipher2 = veCipher2 andalso
    idpVeCipher1 = #cipherVE(sp1Rec1) andalso
    idpVeCipher2 = #cipherVE(sp2Rec1) andalso
    idpVeCipher2 = #cipherVE(sp2Rec2);
val idpVePub1 = #vePubKey(idpRec1)
val idpVePub2 = #vePubKey(idpRec2)
val idpVePubOK = idpVePub1 = #pub(#veKeys(userRec1)) andalso
    idpVePub2 = #pub(#veKeys(userRec2)) ;
val idpCondOK = #conditions1(#idpConditions(idpRec1)) = cond1_1 andalso
    #conditions1(#idpConditions(idpRec2)) = cond1_2 andalso
    #conditions(sp1Rec1) = cond1_1 andalso #conditions(sp1Rec2) = cond1_2
val PKVESuccessBE = PredAllArcs(
    fn a => case ArcToBE a of
    Bind.PE'IDP_GENERATES_STORES_PSEUDONYM_AND_SEND_TO_USER (1, {...} ) => true
    | _ => false);
val PKVESuccess = length PKVESuccessBE = session
val idpPseudo1 = #pseudo(idpRec1)
val idpPseudo2 = #pseudo(idpRec2)
val pseudo1 = #pseudo(userRec1)
val pseudo2 = #pseudo(userRec2);
val idpGenerateCorrectPseudoOK1 =
    valOf(Int.fromString(substring(idpPseudo1, 6, (String.size(idpPseudo1) - 6)))) >0 andalso
    substring(idpPseudo1, 0, 6) = "pseudo";
val idpGenerateCorrectPseudoOK2 =
    valOf(Int.fromString(substring(idpPseudo1, 6, (String.size(idpPseudo2) - 6))))) >0 andalso
    substring(idpPseudo2, 0, 6) = "pseudo";
val pseudoOK = idpGenerateCorrectPseudoOK1 andalso idpGenerateCorrectPseudoOK2 andalso
    idpPseudo1 = pseudo1 andalso idpPseudo2 = pseudo2
val property2 = idpEVCipherOK andalso idpVePubOK andalso idpCondOK
    andalso PKVESuccess andalso pseudoOK;
if property2 then TextIO.output (os, "Property 2 is SATISFIED") else
    TextIO.output (os, "Property 2 is NOT SATISFIED");
TextIO.closeOut os;

Table 12: Session-data and state-space analysis for Property-2
– state3-3: the IDP, and SP1 must receive the correct UCHVE encryption of the VE private key as generated by Module2
– state3-4: the verification of the TPM proof of correct Module2 execution must not fail (again, we do not consider any malicious behaviour from the user yet at this point).

The queries used to verify this property are shown in Table 13.

5.4 Property-4

Property-4 establishes a correct MC stage. Assuming that Property1 holds, at the end of a MC stage, the protocol should be at the following states:

– state4-1: the IDP must have an open authenticated session with the user
– state4-2: the IDP, and SP2 must receive the correct UCHVE encryption of the VE private key as generated by Module2
– state4-3: the verification of the TPM proof of correct Module2 execution must not fail (no malicious behaviour)

The queries used to verify this property are shown in Table 14.

5.5 Property-5

Property5 establishes a correct multiple-session ending property. At the end of executing multiple PIEMC sessions (in our model, we parameterize the model to execute two sessions as indicated by the session parameter in line 56 of Table 2), the protocol should be at the following state:

– state1: given an interaction between a user and SP1 (in session 1 and session 2), the values of the condition string that SP1 has should be the same as the ones that the user, IdP, SP1 have within the same session,
– state2: given an interaction between a user and SP2 (in session 1 and session 2), the values of the condition string that SP2 has should be the same as the ones that the user, IdP, SP2 have within the same session,
– state3: regardless of the session number, the condition string value between SP1 and SP2 should always be different,
– the user-generated one-time VE key pair (K\textsubscript{PUB VE} and K\textsubscript{PRIV VE}) must be different from session 1 to session 2
– state4: the one-time public VE key (K\textsubscript{PUB VE}) that the IDP receives must be consistent with the one that the user has in both sessions.
– state5: the one-time public VE key (K\textsubscript{PUB VE}) value that the IDP receives must be different from one session to another (to represent uniqueness and the one-time property of the keys)
– state6: the value of the CIPHER\_VE\_PII, and the CIPHER\_UCHVE\_KVE must be different from one session to another
– state7: the pseudonym that the IdP generates for the user must be different from one session to another
```scala
val os = TextIO.openOut("property3.txt")
val userRec1 = readUserRecord("user_sess1.txt")
val userRec2 = readUserRecord("user_sess2.txt")
val sp1Rec1 = readSPRecord("sp1_sess1.txt")
val sp1Rec2 = readSPRecord("sp1_sess2.txt")
val idpRec1 = readIDPRecord("idp_sess1.txt")
val idpRec2 = readIDPRecord("idp_sess2.txt")
val cond1_1 = #conditions(#conditions(userRec1))
val cond1_2 = #conditions(#conditions(userRec2))
val condOK = cond1_1 = #conditions(sp1Rec1) andalso cond1_2 = #conditions(sp1Rec2) andalso
#conditions(#idpConditions(idpRec1)) = cond1_1 andalso
#conditions(#idpConditions(idpRec2)) = cond1_2
val aikPub1 = #pubAIK(#tpmKeys(userRec1))
val aikPub2 = #pubAIK(#tpmKeys(userRec2))
val aikOK = aikPub1 = #aikVerifyKey(#verifyKeys(idpRec1)) andalso
aikPub2 = #aikVerifyKey(#verifyKeys(idpRec2))
val sessionSignKey1 = #sessionSignKey(userRec1)
val sessionSignKey2 = #sessionSignKey(userRec2)
val sessionKeyOK = sessionSignKey1 = #sessionVerifyKey(#verifyKeys(idpRec1)) andalso
sessionSignKey2 = #sessionVerifyKey(#verifyKeys(idpRec2))
val ibepreEncMarks = SearchNodes(
EntireGraph,
fn n => length (Mark.KE'TPM_GENERATED_CIPHER_UCHVE_KVE 1 n) > 0,
NoLimit,
fn n => n,
[]) op : ;
val uchve1 = List.nth (Mark.KE'TPM_GENERATED_CIPHER_UCHVE_KVE 1 17, 0)
val uchve2 = List.nth (Mark.KE'TPM_GENERATED_CIPHER_UCHVE_KVE 1 55, 0)
val uchveOK = uchve1 = #cipherUCHVE(sp1Rec1) andalso uchve2 = #cipherUCHVE(sp1Rec2) andalso
uchve1 = #cipherUCHVE1(idpRec1) andalso uchve2 = #cipherUCHVE1(idpRec2);
val DAABE = PredAllArcs(
  fn a => case ArcToBE a of
    Bind.KE'IDP_VERIFIES_STORES_AIK_PUB_KEY_SESSION_KEY (1, {result=x, ...} ) => x=true
| _ => false);
val tpmVerifyMarks = SearchNodes(
EntireGraph,
fn n => length (Mark.KE'TPM_PROOF_VERIFICATION_RESULT 1 n) > 0,
NoLimit,
fn n => n,
[]) op : ;
val tpmResult1 = List.nth (Mark.KE'TPM_PROOF_VERIFICATION_RESULT 1 20, 0)
val tpmResult2 = List.nth (Mark.KE'TPM_PROOF_VERIFICATION_RESULT 1 59, 0)
val tpmModule1OK = tpmResult1 andalso tpmResult2 andalso length tpmVerifyMarks = session
val property3 = condOK andalso sessionKeyOK andalso uchveOK andalso DAASuccess
andalso tpmModule1OK;
if property3 then
TextIO.output (os, "Property 3 is SATISFIED") else
TextIO.output (os, "Property3 is NOT SATISFIED");
TextIO.closeOut os;
```

Table 13: Session-data and state-space analysis for Property-3
val os = TextIO.openOut("property4.txt")

fun verifyRenc(renc1:CIPHER_UCHVE_KVE, enc:CIPHER_UCHVE_KVE, cond:LABEL) = 
  #message(renc1) = #message(enc) andalso 
  #label(renc1) = cond;

val userRec1 = readUserRecord("user_sess1.txt")
val userRec2 = readUserRecord("user_sess2.txt")
val sp2Rec1 = readSPRecord("sp2_sess1.txt")
val sp2Rec2 = readSPRecord("sp2_sess2.txt")
val idpRec1 = readIDPRecord("idp_sess1.txt")
val idpRec2 = readIDPRecord("idp_sess2.txt")

val cond2_1 = #conditions2(#conditions(userRec1))
val cond2_2 = #conditions2(#conditions(userRec2))
val condOK = cond2_1 = #conditions(sp2Rec1) andalso cond2_2 = #conditions(sp2Rec2) andalso 
  #conditions2(#idpConditions(idpRec1)) = cond2_1 andalso 
  #conditions2(#idpConditions(idpRec2)) = cond2_2

val openAuthMarks = SearchNodes(
  EntireGraph, 
  fn n => length (Mark.MC'OPEN_AUTHENTICATED_SESSION_EXISTS 1 n) > 0, 
  NoLimit, 
  fn n => n, 
  [1, 
   op :: ]); 
val openAuthResult1 = List.nth (Mark.MC'OPEN_AUTHENTICATED_SESSION_EXISTS 1 30, 0)
val openAuthResult2 = List.nth (Mark.MC'OPEN_AUTHENTICATED_SESSION_EXISTS 1 68, 0)
val openAuthOK = openAuthResult1 andalso openAuthResult2 andalso length openAuthMarks = session

val idpRenc1 = #cipherUCHVE2(idpRec1)
val idpRenc2 = #cipherUCHVE2(idpRec2)
val idpUchve1 = #cipherUCHVE1(idpRec1)
val idpUchve2 = #cipherUCHVE1(idpRec2)
val idpRencOK = verifyRenc(idpRenc1, idpUchve1, cond2_1) andalso verifyRenc(idpRenc2, idpUchve2, cond2_2)
val idpOK = idpRenc1 = #cipherUCHVE(sp2Rec1) andalso idpRenc2 = #cipherUCHVE(sp2Rec2)
val property4 = condOK andalso openAuthOK andalso idpRencOK andalso idpOK;

if property4 then TextIO.output (os, "Property 4 is SATISFIED")
else TextIO.output (os, "Property 4 is NOT SATISFIED");
TextIO.closeOut os;

Table 14: Session-data and state-space analysis for Property-4
To verify this property, we use the session-data analysis approach as shown in Table 15. As explained in Section 4.2, at the end of each session, we archive the particular session data into a file for each entity. These files act like a ‘database’ that can be used for analysis. Line 1-10 in Table 15 show the process of reading the user, IdP, SP1, and SP2 session data files into a set of variables, each with their respective CPN colour type.

Line 12-36 are the core session-data analysis. Line 12-15 and line 17-20 verify if state1 and state2 are achieved respectively. Line 22-23 verify the achievement of state3. Line 25-26 verify if state4 and state5 are satisfied, while line 28-34 verify the fulfillment of state6. Line 36 verifies the satisfaction of state7. Finally, in order to verify if Property5 has been satisfied, we need to make sure that all state1 – state6 are satisfied. This is verified at line 38-39. Figure 10 shows the result of executing the Table 15 query confirming the satisfaction of Property5.

```
val sameCond1 = true : bool
val sameCond2 = true : bool
val diffCond = true : bool
val condConsistent = true : bool
val oneTimeVEKeys = true : bool
val oneTimeVEPub = true : bool
val oneTimeVECipher = true : bool
val oneTimeUdhwe = true : bool
val oneTimePseudo = true : bool
val property5 = true : bool
val it = (): unit
val prop = condConsistent and also oneTimeVEKeys and also oneTimeVEPub and also oneTimeVECipher and also oneTimeUdhwe and also oneTimePseudo;
```

Fig. 10: Result of executing Property5 Query

**Property-6** Property-6 establishes the correct modeling of the revocation stage. Assuming that SP1 genuinely claims the fulfillment of the conditions, at the end of the revocation stage, a PII must be revealed.

The queries used to verify this property are shown in Table 16.

6 Verification of the PIEMC Protocol

In Section 4.2, we have described the model of PIEMC. In this section, we detail our approach to verifying security properties of the PIEMC protocol. Due to many similarities in verifying the security properties for both the PIEMC protocol with and without trusted ARM, in this paper, we pick the security properties for the PIEMC protocol without the trusted ARM. The main reason
val os = TextIO.openOut("property5.txt");
val userRec1 = readUserRecord("user_sess1.txt");
val userRec2 = readUserRecord("user_sess2.txt");
val sp2Rec1 = readSPRecord("sp2_sess1.txt");
val sp2Rec2 = readSPRecord("sp2_sess2.txt");
val sp1Rec1 = readSPRecord("sp1_sess1.txt");
val sp1Rec2 = readSPRecord("sp1_sess2.txt");
val idpRec1 = readIDPRecord("idp_sess1.txt");
val idpRec2 = readIDPRecord("idp_sess2.txt");

val sameCond1 = #conditions1(#conditions(userRec1)) = #conditions1(#idpConditions(idpRec1))
andalso #conditions1(#conditions(userRec1))= #conditions1(#conditions(userRec2)) andalso
#conditions1(#idpConditions(idpRec1)) = #idpConditions(idpRec2))
andalso #conditions1(#conditions(userRec2)) = #conditions1(#idpConditions(idpRec2));

val sameCond2 = #conditions2(#conditions(userRec1)) = #conditions2(#idpConditions(idpRec1))
andalso #conditions2(#conditions(userRec1))= #conditions2(#conditions(userRec2)) andalso
#conditions2(#idpConditions(idpRec1)) = #idpConditions(idpRec2))
andalso #conditions2(#conditions(userRec2)) = #conditions2(#idpConditions(idpRec2));

val diffCond = #conditions1(#conditions(userRec1)) <> #conditions2(#conditions(userRec1));
val condConsistent = sameCond1 andalso sameCond2 andalso diffCond;

val oneTimeVEKeys = #veKeys(userRec1) <> #veKeys(userRec2);
val oneTimeVEPub = #vePubKey(idpRec1) <> #vePubKey(idpRec2);

val oneTimeVeCipher = #cipherVE(sp1Rec1) <> #cipherVE(sp1Rec2) andalso
#cipherVE(sp2Rec1) <> #cipherVE(sp2Rec2)

val oneTimeUchve = #cipherUCHVE(sp1Rec1) <> #cipherUCHVE(sp1Rec2) andalso
#cipherUCHVE(sp2Rec1) <> #cipherUCHVE(sp2Rec2) andalso
#cipherUCHVE(sp1Rec2) <> #cipherUCHVE(sp1Rec2) andalso
#cipherUCHVE(sp2Rec1) <> #cipherUCHVE(sp2Rec2)

val oneTimePseudo = #pseudo(userRec1) <> #pseudo(userRec2)

val property5 = condConsistent andalso oneTimeVEKeys andalso oneTimeVEPub andalso
oneTimeVeCipher andalso oneTimeUchve andalso oneTimePseudo;

if property5 then TextIO.output(os, "Property 5 is SATISFIED") else
TextIO.output(os, "Property 5 is NOT SATISFIED");

TextIO.closeOut os;

Table 15: Session-data Analysis for Property5
Table 16: Session-data and state-space analysis for Property-6
is because the model description explained in Section 4 is mostly based on the one without trusted ARM. We have performed the security property verification for both variants of the protocol and we notice that the approach used is very similar.

In general, we use two main approaches in verifying security properties: session data analysis and state-space analysis.

**Session-data Analysis** As described in Section 4.2, at the end of each session, we create a new file which stores a session’s data. This data can then be used for the purpose of verifying if some security properties are achieved by using the existing SML and CPN ML functions. In addition, session data is very useful to verify the protocol correctness.

**State-space Analysis** State-space analysis is a very powerful tool to verify security properties of the PIEMC protocol. Our approach in using state-space analysis is normally to combine it with session-data analysis in a multi-step property verification approach: since security properties can be complex, it is necessary to ‘prove’ their satisfaction by translating them into a series of statements that can be verified through a series of state-space and session-data queries.

The verification of PIEMCP are carried out in two stages: the basic behaviour verification and security behaviour verification. The basic behaviour verification is performed through standard state space analysis. It includes analysis of proper session termination, deadlock freedom, livelock freedom, and absence of unexpected dead transitions. The security behaviour verification is the focus of the paper, and is performed through advanced state space analysis and session data analysis. Furthermore, verifying the security behaviour of PIEMCP is complicated due to the numerous avenues in which attackers could break the security protection provided by the protocol. Due to the complexity of the protocol, the potential numerous undetected weak points may lead the protocol to insecure states.

We propose to scope the verification of the security behaviour of PIEMCP within a set of plausible known attack scenarios. The result of such a verification is the assurance that the desired security behaviour are achieved within a set of known attack scenarios. As attacks are parameterised in the model, new types of attack scenarios can be added to the existing model without requiring major changes or new model to be developed.

We define a protocol to be secure if the set of security properties hold in both the presence of attack models and the absence of attack models. This is especially true in the case of PESPs whose main service (privacy) is in itself already a security behaviour. When no attacks are modeled, we expect the security behaviours to be fulfilled; when attacks are included, we expect the protocol to either detect it (and therefore stops), or be immune from those attacks. In this section, we explain our approach through the verification of PIEMCP-NT.

For simplicity, we consider a minimum full protocol execution. The PIEMCP CPN model is parameterised to execute two escrow sessions sequentially, followed by one revocation session. Note that it is possible for both the escrow
and revocation session to run in parallel, however, modeling such a concurrency does not capture any additional behaviours of the protocol as these two sessions are distinct, i.e. they do not interfere with each other. The state space generated from the above in the absence of attack behaviour contains 147 nodes and 226 arcs. Next, the CPN model is parameterised to include a number of plausible known attacks, resulting a set of parameterised CPN models. Each of these models is executed to generate the state space for analysis of certain security properties. Below, we define five security properties and discuss how they are verified using session data analysis and state space analysis.

Approach Generally, we use the following approach to verifying security properties:

1. translate the security properties into a series of statements that can be verified using either the session-data analysis or state-space analysis,
2. write the necessary queries to verify the statements under the assumption that all entities behave honestly (that is, no attack behaviours are included).
3. If the queries confirm the satisfaction of a security property, proceeds to verify the property with attack behaviours. Else, stop.
4. Decide on a set of suitable attacks that may compromise the satisfaction of the security property.
5. If necessary, modify the statements that are suitable to verify the security property in the presence of the attacks
6. Write queries to verify the security property in the presence of the modeled attack(s).

As we explained in the introduction (Section 1), exhaustive identification of all possible attacks and the corresponding exhaustive verification of the protocol behaviour in response to those attacks is impossible. Consequently, it is important that our approach to verifying a protocol’s security property needs to be expandable. Our approach proposed above is expandable, that is, step 4-6 can be repeatedly performed to cover as many additional attack behaviours as necessary. With a parameterization approach to modeling attack behaviour, it is possible that we do not have to ‘break’ the existing model; rather, new attack behaviour can simply be added to the model such that the existing model for the protocol, attack behaviours, and their corresponding state-space or session-data queries are retained and still usable.

To illustrate our approach to verifying these security properties, we explain in detail our approach to verifying security properties as detailed in Section 3.3.

Property 1 (Multiple Conditions). At the end of every escrow session, the PII escrowed must be bound to multiple sets of alternative conditions, each to be used with a different SP.

To verify this property, we need to verify that the UCHVE ciphertexts produced at the KE and MC stages encrypt the same message (the VE public key) but
under different conditions. Therefore, in the absence of attacks, this property can be verified by doing a simple session-data analysis.

The queries that we can use to verify the above property are shown in Table 17.

```
val sp1Rec1 = readSPRecord("sp1_sess1.txt")
val sp1Rec2 = readSPRecord("sp1_sess2.txt")
val sp2Rec1 = readSPRecord("sp2_sess1.txt")
val sp2Rec2 = readSPRecord("sp2_sess2.txt")
val cipherUCHVE11 = #cipherUCHVE(sp1Rec1)
val cipherUCHVE21 = #cipherUCHVE(sp2Rec1)
val cipherUCHVE12 = #cipherUCHVE(sp1Rec2)
val cipherUCHVE22 = #cipherUCHVE(sp2Rec2)
val messageSame = #message(cipherUCHVE11) = #message(cipherUCHVE21) andalso 
  #message(cipherUCHVE12) = #message(cipherUCHVE22);
val differentCond = #label(cipherUCHVE11) <> #label(cipherUCHVE21) andalso 
  #label(cipherUCHVE12) <> #label(cipherUCHVE22);
val multipleConditions = messageSame andalso differentCond;
```

Table 17: State-space and session data analysis for the Multiple Conditions property - normal

However, when SP1 and SP2 collude and attempt to fool the user to accept the same condition string (captured in the attack parameter SP_ATTACK7), this property may be violated. Our initial verification shows that this property does not hold when this attack parameter is switched on. We found that this is due to a missing operation in the original protocol design which requires the user to verify the one-time property of the condition string before being used. A simple function to capture such a one-time condition string property verification is therefore defined and added to the protocol design. With this function included, such an attack is detected and the protocol stops (verified by checking the protocol dead marking). Therefore, this property holds in the updated protocol definition.

Property 2 (Zero-knowledge). Prior to conditions fulfillment, IdPs, SPs, and referees must not learn the value of the user’s PII but at the same time be convinced that its encryption is correct; when conditions are fulfilled, the revealed PII must indeed be a correct certified PII

Without considering attacks, the first verification part can simply be performed by examining session-data analysis to make sure that the IdP, SP1, and SP2 only possess the ciphertext forms of the VE private key and PII - not their plain-text form. Recall that a message field in a record representing a ciphertext is treated as unreadable. The second verification part (after conditions fulfillment) can be performed using a simple state-space analysis. Our verification process confirms that this property is satisfied when no attack is included. The session data and state-space analysis used is shown in Table 18.
val os = TextIO.openOut ("zero-knowledge.txt");
TextIO.output (os, "Verifying the Zero-knowledge property.\n");
val userRec2 = readUserRecord("user_sess2.txt");
val sp2Rec1 = readSPRecord("sp2_sess1.txt");
val sp2Rec2 = readSPRecord("sp2_sess2.txt");
val sp1Rec1 = readSPRecord("sp1_sess1.txt");
val sp1Rec2 = readSPRecord("sp1_sess2.txt");
val startRevokeMarkList = SearchNodes(
EntireGraph,
fn n => length (Mark.REVOCATION\'-REVOCATION_CONDITIONS_FULFILLED 1 n) > 0,
NoLimit, fn n => n, [], op :: );
val startRevokeMark = List.nth (startRevokeMarkList, 0)
val revealedMarks = SearchNodes(
EntireGraph,
fn n => length (Mark.PIEMC\'-RECOVERED_USER_PII 1 n) > 0 orelse
length (Mark.REVOCATION\'-RECOVERED_USER_PII 1 n) > 0 orelse
length (Mark.REVOCATION\'-VS_PRIVATE_KEY 1 n) > 0 orelse
length (Mark.REVOCATION\'-REF_RECOVERED_VE_PRIVATE_KEY 1 n) > 0,
NoLimit, fn n => n, [], op :: );
val wrongMarks = SearchNodes(
revealedMarks,
fn n => not (Reachable (startRevokeMark, n)),
NoLimit, fn n => n, [], op :: );
val noLearn = length wrongMarks = 0
val correctPII = #pii(#cert(userRec1)) = #message(#cipherVE(sp1Rec1)) andalso
#pii(#cert(userRec1)) = #message(#cipherVE(sp2Rec1)) andalso
#pii(#cert(userRec2)) = #message(#cipherVE(sp1Rec2)) andalso
#pii(#cert(userRec2)) = #message(#cipherVE(sp2Rec2)) ;
val correctUCHVE = #priv(#veKeys(userRec1)) = #message(#cipherUCHVE(sp1Rec1)) andalso
#priv(#veKeys(userRec1)) = #message(#cipherUCHVE(sp2Rec1)) andalso
#priv(#veKeys(userRec2)) = #message(#cipherUCHVE(sp1Rec2)) andalso
#priv(#veKeys(userRec2)) = #message(#cipherUCHVE(sp2Rec2));
if correctPII andalso correctUCHVE then
TextIO.output (os, "At the end of each session, SPs have correct ciphertexts.\n") else
TextIO.output (os, "At the end of each session, SPs does not have correct ciphertexts.\n");
TextIO.output (os, "Find marking where user PII is revoked, and compare the revoked value.\n");
val revealedMark = SearchNodes(
EntireGraph,
fn n => length (Mark.REVOCATION\'-RECOVERED_USER_PII 1 n) > 0,
NoLimit, fn n => n, [], op :: );
val revealedNode = SearchNodes(
EntireGraph,
fn n => length (Mark.PIEMC\'-RECOVERED_USER_PII 1 n) > 0 orelse
length (Mark.REVOCATION\'-RECOVERED_USER_PII 1 n) > 0 orelse
length (Mark.REVOCATION\'-VS_PRIVATE_KEY 1 n) > 0 orelse
length (Mark.REVOCATION\'-REF_RECOVERED_VE_PRIVATE_KEY 1 n) > 0,
NoLimit, fn n => n, [], op :: );
val correctRevealed = if length revealedNode>0 andalso
List.nth (Mark.REVOCATION\'-RECOVERED_USER_PII 1 mark1, 0) = #pii(#cert(userRec1)) andalso
List.nth (Mark.REVOCATION\'-RECOVERED_USER_PII 1 mark2, 0) = #pii(#cert(userRec1)) andalso
List.nth (Mark.REVOCATION\'-RECOVERED_USER_PII 1 mark3, 0) = #pii(#cert(userRec1)) andalso
List.nth (Mark.REVOCATION\'-RECOVERED_USER_PII 1 mark4, 0) = #pii(#cert(userRec1))
then true else false;
val failProofList = SearchNodes(
EntireGraph,
fn n => length (Mark.PE\'-PII_VE_CIPHER_RESULT 1 n) > 0 andalso
not (List.nth (Mark.PE\'-PII_VE_CIPHER_RESULT 1 n, 0)),
NoLimit, fn n => n, [], op :: );
if noLearn andalso correctPII andalso correctUCHVE andalso correctRevealed
andalso proofSuccess then
TextIO.output (os, "The Authenticated PII property is SATISFIED.\n") else
TextIO.output (os, "The Authenticated PII property is NOT SATISFIED.\n");
TextIO.closeOut os;

Table 18: State-space and session data analysis for the zero-knowledge property
Next, we determine the attack scenarios that may violate this security behaviour. One plausible attack is when the SP1 or SP2 colludes with the referees. Together, they may be able to decrypt the UCHVE ciphertext to reveal the VE private key and subsequently reveal the PII. This attack is captured by the REF.Attack2 parameter. In particular, we need to verify that

- prior to conditions fulfillment, the IdP, SP, and referees do not learn the value of the user’s PII. This can be verified by checking if the places which indicate the K_PRIV.VE or PII value in cleartext have empty token prior to the revocation stage. These places include:
  - PI.EMC.RECOVERED.USER.PII
  - REVOCATION.RECOVERED.USER.PII
  - REVOCATION.VE.PRIVATE.KEY, and
  - REVOCATION.REF.RECOVERED.VE.PRIVATE.KEY
- the VE ciphertext that SP1 and SP2 receive must correctly encrypt user certified PII,
- the UCHVE ciphertexts that SP1 and SP2 receive must correctly encrypt the corresponding VE private key that the user generated per session,
- all of the above ciphertexts are encrypted under the correct labels (this has been verified from Property 2, Property 3, and Property 4),
- the place PE.PII.VE.CIPHER_RESULT, KE.TPM.PROOF_VERIFICATION_RESULT, and MC.TPM.PROOF_VERIFICATION_RESULT must never contain a value of ‘false’ - this is to indicate that all zero-knowledge protocol and TPM proofs are accepted by the IdP as a convincing evidence of correct encryptions.
- After conditions fulfillment (that is, when a revocation of the PII is successful), the revoked PII must be the same as the one stored in the user certificate.

Having switched on this attack, we executed the same queries (with minor modification) to confirm that this property holds in the presence of REF.Attack2 given that there are only \( t - 1 \) out of \( t \) dishonest designated referees. When there are \( t \) dishonest referees, this property is violated. The queries used is detailed in Table 19

Property 3 (Enforceable Conditions). A user’s PII should never be revealed before all designated referees agree that the cryptographically bounded conditions are satisfied.

We translate this security property into the following statements:

- When the associated conditions are satisfied (that is, the condActually parameter is set to ‘true’), there must be
  - a marking where the place PI.REVOCATION.REF1_COND_FULFILLMENT and PI.REVOCATION.REF2_COND_FULFILLMENT (the designated referees) are populated with a token of value ‘true’ simultaneously. We call this marking M1 - this marking implies that all designated referees have agreed that the revocation conditions are satisfied. The decision of non-designated referee is irrelevant),
val os = TextIO.openOut("zk_attack.txt");
val userRec1 = readUserRecord("user_sess1.txt");
val userRec2 = readUserRecord("user_sess2.txt");
val sp2Rec1 = readSPRecord("sp2_sess1.txt");
val sp2Rec2 = readSPRecord("sp2_sess2.txt");
val sp1Rec1 = readSPRecord("sp1_sess1.txt");
val sp1Rec2 = readSPRecord("sp1_sess2.txt");
val startRevokeMarkList = SearchNodes(EntireGraph, fn n => length (Mark.REVOCATION'REVOCATION_CONDITIONS_FULFILLED 1 n) > 0, NoLimit, fn n => n, [], op ::);
val startRevokeMark = List.nth (startRevokeMarkList, 0)
val revealedMarks = SearchNodes(EntireGraph, fn n => length (Mark.PIEMC'RECOVERED_USER_PII 1 n) > 0 orelse length (Mark.REVOCATION'VE_PRIVATE_KEY 1 n) > 0 orelse length (Mark.REVOCATION'REF_RECOVERED_VE_PRIVATE_KEY 1 n) > 0, NoLimit, fn n => n, [], op ::);
val wrongMarks = SearchNodes(revealedMarks, fn n => not (Reachable (startRevokeMark, n)), NoLimit, fn n => n, [], op ::);
val noLearn = length wrongMarks = 0
val correctPII = #pii(#cert(userRec1)) = #message(#cipherVE(sp1Rec1)) andalso #pii(#cert(userRec1)) = #message(#cipherVE(sp2Rec1)) andalso #pii(#cert(userRec2)) = #message(#cipherVE(sp1Rec2)) andalso #pii(#cert(userRec2)) = #message(#cipherVE(sp2Rec2));
val correctUCHVE = #priv(#veKeys(userRec1)) = #message(#cipherUCHVE(sp1Rec1)) andalso #priv(#veKeys(userRec1)) = #message(#cipherUCHVE(sp2Rec1)) andalso #priv(#veKeys(userRec2)) = #message(#cipherUCHVE(sp1Rec2)) andalso #priv(#veKeys(userRec2)) = #message(#cipherUCHVE(sp2Rec2));
if correctPII andalso correctUCHVE then TextIO.output (os, "At the end of each session, SPs have correct ciphertexts.\n\n") else TextIO.output (os, "At the end of each session, SPs does not have correct ciphertexts.\n\n");
if correctPII andalso correctUCHVE then TextIO.output (os, "Find marking where user PII is revoked, and compare the revoked value.\n\n");
val revealedNode = SearchNodes(EntireGraph, fn n => length (Mark.PIEMC'RECOVERED_USER_PII 1 n) > 0 orelse length (Mark.REVOCATION'VE_PRIVATE_KEY 1 n) > 0 orelse length (Mark.REVOCATION'REF_RECOVERED_VE_PRIVATE_KEY 1 n) > 0, NoLimit, fn n => n, [], op ::);
val refLearn = SearchNodes(EntireGraph, fn n => length (Mark.REVOCATION'REF_RECOVERED_VE_PRIVATE_KEY 1 n) > 0, NoLimit, fn n => n, [], op ::);
val failProofList = SearchNodes(EntireGraph, fn n => length (Mark.PIEMC'PII_VE_CIPHER_RESULT 1 n) > 0 andalso not (List.nth (Mark.PIEMC'PII_VE_CIPHER_RESULT 1 n, 0)), NoLimit, fn n => n, [], op ::);

|
| Table 19: State-space and session data analysis for the zero-knowledge property with REF_ATTACK2 |

---
• another marking indicating that the user’s PII is revealed (indicated by a token in the place PI.REVOCATION’RECOVERED_USER_PII) - we call this marking M2, and
• M2 must only exist if there is M1, and M1 must happen before M2.
  – When the associated conditions are not satisfied (that is, the condActually parameter is set to ‘false’), there must be
    • a marking where the place PI.REVOCATION’REF1_COND_FULFILLMENT and PI.REVOCATION’REF2_COND_FULFILLMENT (the designated referees) are populated with a token of value ‘false’ simultaneously. We call this marking M1’ - this marking implies that all designated referees have decided that the revocation conditions are not satisfied.
    • M2 should NOT exist.

The above statements are translated into a series of queries as shown in Table 20.

When condActually=true, we have line 3-16 in Table 20 which verify if M1 exists, and if so, if it is valid. Similarly, line 17-24 verify the existence and validity of M2. Line 25-33 verify if M1 happens before M2. The enforceable condition property is fulfilled when all of the above verification succeed (line 34).

When condActually=false, then we have line 39-49 verifying the existence and validity of M1’. Line 50-55 verify the non-existence of M2. Finally, the enforceable conditions property is satisfied when the above two verifications succeed (line 56).

The results for both queries are shown in Figure 11. They confirm the fulfillment of the enforceable conditions property.

Next, we need to decide on a set of relevant attacks that may compromise the fulfillment of this property. We decide that these attacks include SP_ATTACK3, SP_ATTACK4, REF_ATTACK1, and REF_ATTACK2. When these attacks are included, we need to show that:

– if SP_ATTACK3 is enabled, then we should expect the referees to still agree of conditions fulfillment, but decryption of the UCHVE pieces should fail. Therefore, revocation stage will stop prematurely: we should expect M1, but there should be no M2, and we should expect empty token at the place REVOCATION’UCHVE_PIECE_DECRYPT_SUCCESS.

– if SP_ATTACK4 is enabled, then it is logical to also enable REF_ATTACK1 (to mimic collusion between SP/IDP and referees). In this case, we expect two malicious referees to still be able to decrypt the UCHVE piece, however, the one honest referee does not. Therefore, M1 and M2 should not exist as well and the place REVOCATION’UCHVE_PIECE_DECRYPT_SUCCESS should only contain 1 token (as a result of decryption by malicious referee - honest referee won’t decrypt, and non-designatedRef returns failed decryption token).

– in both scenario above, REF_ATTACK2 can be enabled simultaneously to simulate referees trying to recover K_PRIV_VE. In this case, we need to verify that the place PI.REF_RECOVERED_VE_PRIVATE_KEY should never have any empty token (maximum bound is 0).
State-space query when \( \text{condActually=true} \):

```plaintext
val m1Nodes = SearchNodes(EntireGraph,
fn n => length (Mark.REVOCATION*REF1_COND_FULFILLMENT 1 n) > 0 andalso
length (Mark.REVOCATION*REF2_COND_FULFILLMENT 1 n) > 0,
NoLimit, fn n => n, [], op ::);
```

```plaintext
val m1Len = length m1Nodes;
val m11 = List.nth (m1Nodes, 0); val m12 = List.nth (m1Nodes, 1);
val m13 = List.nth (m1Nodes, 2); val m14 = List.nth (m1Nodes, 3);
```

```plaintext
val m1Correct = if length m1Nodes>0 andalso
List.nth (Mark.REVOCATION*REF1_COND_FULFILLMENT 1 m11, 0) andalso
List.nth (Mark.REVOCATION*REF2_COND_FULFILLMENT 1 m12, 0) andalso
List.nth (Mark.REVOCATION*REF2_COND_FULFILLMENT 1 m13, 0) andalso
List.nth (Mark.REVOCATION*REF2_COND_FULFILLMENT 1 m14, 0)
then true else false;
```

```plaintext
val m2Nodes = SearchNodes(EntireGraph,
fn n => length (Mark.REVOCATION*RECOVERED_USER_PII 1 n) > 0,
NoLimit, fn n => n, [], op ::);
```

```plaintext
val m2Len = length m2Nodes;
val m21 = List.nth (m2Nodes, 0); val m22 = List.nth (m2Nodes, 1);
val m23 = List.nth (m2Nodes, 2); val m24 = List.nth (m2Nodes, 3);
```

```plaintext
val m2Correct = m2Len > 0;
val ok1 = Reachable (m11, m21) orelse Reachable (m12, m21) orelse
Reachable (m13, m21) orelse Reachable (m14, m21);
val ok2 = Reachable (m11, m22) orelse Reachable (m12, m22) orelse
Reachable (m13, m22) orelse Reachable (m14, m22);
val ok3 = Reachable (m11, m23) orelse Reachable (m12, m23) orelse
Reachable (m13, m23) orelse Reachable (m14, m23);
val ok4 = Reachable (m11, m24) orelse Reachable (m12, m24) orelse
Reachable (m13, m24) orelse Reachable (m14, m24);
```

```plaintext
val precedence = ok1 andalso ok2 andalso ok3 andalso ok4;
val enfCondFulfilled = m1Correct andalso m2Correct andalso precedence;
```

State-space query when \( \text{condActually=false} \):

```plaintext
val m1Nodes = SearchNodes(EntireGraph,
fn n => length (Mark.REVOCATION*REF1_COND_FULFILLMENT 1 n) > 0 andalso
length (Mark.REVOCATION*REF2_COND_FULFILLMENT 1 n) > 0,
NoLimit, fn n => n, [], op ::);
```

```plaintext
val m1Len = length m1Nodes;
val m11 = List.nth (m1Nodes, 0); val m12 = List.nth (m1Nodes, 1);
```

```plaintext
val m1Correct = if length m1Nodes>0 andalso
not (List.nth (Mark.REVOCATION*REF1_COND_FULFILLMENT 1 m11, 0)) andalso
not (List.nth (Mark.REVOCATION*REF2_COND_FULFILLMENT 1 m12, 0))
then true else false;
```

```plaintext
val m2Nodes = SearchNodes(EntireGraph,
fn n => length (Mark.REVOCATION*RECOVERED_USER_PII 1 n) > 0,
NoLimit, fn n => n, [], op ::);
```

```plaintext
val m2Len = length m2Nodes;
```

```plaintext
val m2Correct = m2Len =0;
```

```plaintext
val enfCondFulfilled = m1Correct andalso m2Correct;
```

Table 20: State-space analysis for enforceable conditions property
We now are in the position to write queries to check the above statements. These queries are shown in Table 21.

From Table 21, we can see that the query is divided into two parts: one to show when only SP\_ATTACK3 and REF\_ATTACK2 that are included (line 1-32), and another one when we include only SP\_ATTACK4, REF\_ATTACK1, and REF\_ATTACK2 (collusion between SP1 and malicious referees - line 35-67). The reason for splitting the queries into two parts is because some of the attacks (notably SP\_ATTACK3 and SP\_ATTACK4) are not logically possible to be launched simultaneously.

Finally, we can execute the query to verify if the security property still holds. The result of our query is shown in Figure 12. The result shows that the enforceable condition property still holds even in the presence of the mentioned attacks.

Property 4 (Conditions Abuse Resistant). An SP and an IdP must not be able to fool the user to encrypt the PII, or the VE private key, under a set of conditions different from those originally agreed. Similarly, an SP or IdP must not be able to successfully revoke the user’s PII using conditions different from those originally agreed.

When all parties behave honestly, the verification of this property is straightforward:
WHEN SP_ATTACK3 and REF_ATTACK2 are included:
val m1Nodes = SearchNodes(
EntireGraph, fn n => length (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 n) > 0 andalso
length (Mark.REVOCATION'REF2_COND_FULFILLMENT 1 n) > 0,
NoLimit, fn n => n, [], op ::);
val m1Len = length m1Nodes; val m11 = List.nth (m1Nodes, 0);
val m12 = List.nth (m1Nodes, 1); val m13 = List.nth (m1Nodes, 2);
val m14 = List.nth (m1Nodes, 3); val m15 = List.nth (m1Nodes, 4);
val m16 = List.nth (m1Nodes, 5);
val m1Correct = if length m1Nodes>0 andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m11 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m12 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m13 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m14 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m15 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m16 , 0) then true else false;
val m2Nodes = SearchNodes(
EntireGraph, fn n => length (Mark.REVOCATION'RECOVERED_USER_PII 1 n) > 0,
NoLimit, fn n => n, [], op ::);
val m2Len = length m2Nodes;
val m2Correct = if m2Len=0 then true else false;
val failedDecrypt = UpperInteger (Mark.REVOCATION'UCHVE_PIECE_DECRYPT_SUCCESS 1) = 0;
val refNoRecover = UpperInteger (Mark.REVOCATION'REF_RECOVERED_VE_PRIVATE_KEY 1) = 0;
val enfCondFulfilled = m1Correct andalso m2Correct andalso refNoRecover andalso failedDecrypt;

WHEN SP_ATTACK4, REF_ATTACK1, and REF_ATTACK2 are included:
val os = TextIO.openOut("enforceable_attack.txt");
TextIO.output (os, "Check M1 marking:
");
val m1Nodes = SearchNodes(
EntireGraph, fn n => length (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 n) > 0 andalso
length (Mark.REVOCATION'REF2_COND_FULFILLMENT 1 n) > 0,
NoLimit, fn n => n, [], op ::);
val m1Len = length m1Nodes; val m11 = List.nth (m1Nodes, 0);
val m12 = List.nth (m1Nodes, 1); val m13 = List.nth (m1Nodes, 2);
val m14 = List.nth (m1Nodes, 3); val m15 = List.nth (m1Nodes, 4);
val m16 = List.nth (m1Nodes, 5);
val m1Correct = if length m1Nodes>0 andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m11 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m12 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m13 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m14 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m15 , 0) andalso
List.nth (Mark.REVOCATION'REF1_COND_FULFILLMENT 1 m16 , 0) then true else false;
val m2Nodes = SearchNodes(
EntireGraph, fn n => length (Mark.REVOCATION'RECOVERED_USER_PII 1 n) > 0,
NoLimit, fn n => n, [], op ::);
val m2Len = length m2Nodes;
val m2Correct = if m2Len=0 then true else false;
val failedDecrypt = UpperInteger (Mark.REVOCATION'UCHVE_PIECE_DECRYPT_SUCCESS 1) = 1;
val refNoRecover = UpperInteger (Mark.REVOCATION'REF_RECOVERED_VE_PRIVATE_KEY 1) = 0;
val enfCondFulfilled = m1Correct andalso m2Correct andalso refNoRecover andalso failedDecrypt;

Table 21: State-space analysis for enforceable conditions property with attack behaviours included
– in a session, the label used to encrypt the user PII must be the same between what the user has, and the SP1 and SP2 have,
– in a session, the label used to encrypt the VE one-time private key must be the same between what the user has and SP1,
– in a session, the label used to re-encrypt the VE one-time private key must be the same between what the user has and SP2.

The queries used to verify this property is shown in Table 22.

Nevertheless, there are several types of attacks that could be launched in an attempt to compromise this property. We capture such attacks in USER_ATTACK1, USER_ATTACK4, SP_ATTACK1, SP_ATTACK11, and SP_ATTACK2. Therefore, to verify this property in the presence of such attacks, we need to show that a secure protocol must behave as follows:

– when USER_ATTACK1 or SP_ATTACK1 is included, the PKVE operation should fail and the protocol should terminate,
– when USER_ATTACK4 is included, the TPM Proof verification by the IDP should fail and protocol terminates,
– when SP_ATTACK2 is included, the user must be able to detect it and as a result, prevent the ARM from doing the re-encryption of the VE private key.

The queries used to verify this property in the presence of the detailed attacks are shown in Table 23.

Property 5 (Escrow Session Unlinkability). SPs and IdPs must not be able to link a user from one escrow session to another from the session data gathered.

In the absence of attack model, this property can be verified by showing that that multiple session executions of the escrow session do not leave any session data which can be used to link these sessions to the same user. In other words,
Table 22: State-space and session data analysis for the Conditions-Abuse property - no attack scenarios

```scala
val os = TextIO.openOut("condAbuse.txt")
val userRec1 = readUserRecord("user_sess1.txt")
val userRec2 = readUserRecord("user_sess2.txt")
val idpRec1 = readIDPRecord("idp_sess1.txt")
val idpRec2 = readIDPRecord("idp_sess2.txt")
val sp1Rec1 = readSPRecord("sp1_sess1.txt")
val sp1Rec2 = readSPRecord("sp1_sess2.txt")
val sp2Rec1 = readSPRecord("sp2_sess1.txt")
val sp2Rec2 = readSPRecord("sp2_sess2.txt")
val cond1_1 = #conditions1(#conditions(userRec1))
val cond2_1 = #conditions2(#conditions(userRec1))
val cond1_2 = #conditions1(#conditions(userRec2))
val cond2_2 = #conditions2(#conditions(userRec2))
val idpSP1CondOK = #conditions1(#idpConditions(idpRec1)) = cond1_1 andalso
    #conditions1(#idpConditions(idpRec2)) = cond1_2 andalso
    #conditions(sp1Rec1) = cond1_1 andalso
    #conditions(sp1Rec2) = cond1_2
val idpSP2CondOK =
    cond2_1 = #conditions(sp2Rec1) andalso cond2_2 = #conditions(sp2Rec2) andalso
    #conditions2(#idpConditions(idpRec1)) = cond2_1 andalso
    #conditions2(#idpConditions(idpRec2)) = cond2_2
```

Table 22: State-space and session data analysis for the Conditions-Abuse property - no attack scenarios

Each escrow session data (including VE public key, VE ciphertext, UCHVE ciphertext, conditions, and so on) must be unique to that session only. A simple session-data analysis (as shown in Table 24 confirms that this property is satisfied.

At this point, we do not find any plausible attack scenarios that could violate this property. Due to the non-deterministic nature of the cryptographic primitives used, and due to the zero-knowledge property, the IdPs and SPs will not know the identity of the user, and thus, do not have the knowledge to tell if they are interacting with the same user across multiple escrow sessions. This property may be violated if a user applies the same session data (such as same VE keys, same conditions string) across multiple escrow sessions. Nevertheless, such actions contradict users’ interest (which is to preserve their privacy) and are therefore not considered as a valid ‘attack’.

7 Conclusion

We have shown that CPNs can be used to model and verify complex PESP\s in a scalable and extensible manner. These are two important factors in designing a secure PESP, given that the security behaviour and the attack scenarios involved are application-specific, instead of well-defined as normally encountered in designing cryptographic primitives. We have also demonstrated how security properties of a PESP can be reasoned using state spaces and session data gen-
1 USER_ATTACK1 or SP_ATTACK1
2 TextIO.output (os, "Check behaviour when incorrect conditions used at PE stage:\n\n");
3 val PKVESuccessBE = PredAllArcs(
4   fn a => case ArcToBE a of
5       Bind.PE'IDP_GENERATES_STORES_PSEUDONYM_AND_SEND_TO_USER (1, {...} ) => true
6           | _ => false);
7 val correct1 = length PKVESuccessBE = 0;
8 if correct1 then TextIO.output (os, "Protocol behaves correctly when incorrect
9     conditions used at PE stage.\n\n") else
10   TextIO.output(os, "Protocol behaves unexpectedly when incorrect
11     conditions used at PE stage.\n\n");
12 TextIO.closeOut os;

13 USER_ATTACK4 or SP_ATTACK11
14 TextIO.output (os, "Check behaviour when incorrect conditions used during KE stage:\n\n");
15 val TPMProofSuccessBE = PredAllArcs(
16   fn a => case ArcToBE a of
17       Bind.KE'IDP_VERIFIES_MODULE_2_TPM_PROOF (1, {result=x,...} ) => x=true
18           | _ => false);
19 val expectedTPMBE1 = length TPMProofSuccessBE = 0;
20 if expectedTPMBE1 then TextIO.output (os, "Protocol behaves correctly when
21     incorrect conditions used at KE stage.\n\n") else
22   TextIO.output(os, "Protocol behaves unexpectedly when incorrect conditions
23     used at KE stage.\n\n");
24 TextIO.closeOut os;

25 SP_ATTACK2
26 TextIO.output (os, "Check behaviour when incorrect conditions used during MC stage:\n\n");
27 val TPMProofSuccessBE2 = PredAllArcs(
28   fn a => case ArcToBE a of
29       Bind.MC'IDP_VERIFIES_MODULE_2_TPM_PROOF (1, {result=x,...} ) => x=true
30           | _ => false);
31 val expectedTPMBE2 = length TPMProofSuccessBE = 0;
32 if expectedTPMBE2 then TextIO.output (os, "Protocol behaves correctly when
33     incorrect conditions used at MC stage.\n\n") else
34   TextIO.output(os, "Protocol behaves unexpectedly when incorrect conditions
35     used at MC stage.\n\n");
36 TextIO.closeOut os;

Table 23: State-space and session data analysis for the Conditions-Abuse property with attacks
val os = TextIO.openOut("linkable.txt");

val userRec1 = readUserRecord("user_sess1.txt");
val userRec2 = readUserRecord("user_sess2.txt");
val sp2Rec1 = readSPRecord("sp2_sess1.txt");
val sp2Rec2 = readSPRecord("sp2_sess2.txt");
val sp1Rec1 = readSPRecord("sp1_sess1.txt");
val sp1Rec2 = readSPRecord("sp1_sess2.txt");
val idpRec1 = readIDPRecord("idp_sess1.txt");
val idpRec2 = readIDPRecord("idp_sess2.txt");

val sameCond1 = #conditions1(#conditions(userRec1)) = #conditions1(#idpConditions(idpRec1)) andalso
                   #conditions1(#conditions(userRec1)) = #conditions1(#idpConditions(idpRec2)) andalso
                   #conditions1(#conditions(userRec2)) = #conditions1(#idpConditions(idpRec2));

val sameCond2 = #conditions2(#conditions(userRec1)) = #conditions2(#idpConditions(idpRec1)) andalso
                   #conditions2(#conditions(userRec1)) = #conditions2(#idpConditions(idpRec2));

val diffCond = #conditions1(#conditions(userRec1)) <> #conditions1(#conditions(userRec1));

val oneTimeVEKeys = #veKeys(userRec1) <> #veKeys(userRec2);
val oneTimeVPub = #vePubKey(idpRec1) <> #vePubKey(idpRec2);

val oneTimeVeCipher = #cipherVE(sp1Rec1) <> #cipherVE(sp1Rec2) andalso
                       #cipherVE(sp2Rec1) <> #cipherVE(sp2Rec2);

val oneTimeUchve = #cipherUCHVE(sp1Rec1) <> #cipherUCHVE(sp1Rec2) andalso
                   #cipherUCHVE(sp2Rec1) <> #cipherUCHVE(sp2Rec2);

val oneTimePseudo = #pseudo(userRec1) <> #pseudo(userRec2);

val property5 = condConsistent andalso oneTimeVEKeys andalso oneTimeVPub andalso
                oneTimeVeCipher andalso oneTimeUchve andalso oneTimePseudo;

if property5 then TextIO.output(os, "Linkable property is SATISFIED") else
TextIO.output(os, "Linkable property is NOT SATISFIED");

TextIO.closeOut os;

Table 24: State-space and session data analysis for the Escrow Session Unlinkability property - no attack
erated from execution of the CPN models. This investigation has also led us to a CPN-based modeling and verification approach to PESPs in general.

Future work involves using the model to do a performance analysis of the protocol to assess its efficiency during deployment. We also need to refine and generalize the modeling techniques proposed in this paper (the *cryptographic primitive abstraction, attack parameterization, session data capture*) such that they can be applied to other PESPs. Finally, we envisage the development of a user front-end to simplify and automate as much as possible the tasks required in the modeling and verification of PESPs. The function of such a front-end could be as simple as aiding users with the configuration of the model parameters, to a full-blown automation whereby a user who does not have any knowledge of CPN can generate the required back-end CPN model with only a PESP specification.

**References**

1. of the Australian Privacy Commissioner Australia, O.: National privacy survey: Id theft, id scanning and online privacy concerns are on the rise. (August 2007)
27. van der Aalst, W.: Pi calculus versus petri nets: Let us eat humble pie rather than further inflate the pi hype. BP Trends (May 2005) 1–11