Université de Montréal

Un modèle de validation automatique de mécanismes de sécurisation des communications

par

Fathya Zemmouri

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maîtrise
en informatique

Avril, 2003

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé :

Un modèle de validation automatique de mécanismes de sécurisation des communications

présenté par :

Fathya Zemmouri

a été évalué par un jury composé des personnes suivantes :

Stefan Wolf

président-rapporteur

Peter Kropf

directeur de recherche

Gilbert Babin

codirecteur

Brigitte Jaumard

membre du jury

Mémoire accepté le 26 septembre 2003

Sommaire

Une bonne partie de la cryptanalyse consiste à chercher les failles intrinsèques dans les règles qui définissent les protocoles de sécurité tout en supposant que les algorithmes de cryptographie utilisés dans ces protocoles sont incassables. Ces failles mènent à des attaques par des intrus illégitimes.

Nous proposons dans ce mémoire un modèle de validation automatique de la jointure de plusieurs protocoles de sécurité afin de démontrer la conformité de leur spécification qui prétend couvrir un certain nombre de risques de sécurité dans un contexte de transaction entre plusieurs participants.

Notre modèle de validation est basé sur un langage formel de spécification des protocoles distribués PROMELA et sur un outil de simulation des protocoles concurrents appelé SPIN.

L'une des parties la plus critique dans le modèle de validation est la description exhaustive du comportement d'un intrus dans l'environnement où se déroule la communication à sécuriser. Vu qu'un tel comportement est non déterministe dans n'importe quel espace de problèmes de confiance, un modèle de validation de protocole de sécurité se doit alors de le rendre au mieux déterministe. D'où l'utilisation dans notre modèle des EFSMs (*Extended Finite State Machines*) pour décrire le comportement des intrus.

Nous démontrons le principe de notre modèle de validation par une étude de cas, soit la validation d'une union de mécanismes de sécurité. Ces mécanismes de sécurité choisis pour notre étuse de cas sont : une signature numérique pour fournir l'intégrité des données et la non-répudiation, un certificat numérique pour assurer l'authentification des participants et un algorithme de cryptographie symétrique pour assurer la confidentialité.

Mots-clés : Validation, mécanisme de sécurité, protocole de sécurité, spécification formelle, SecAdvise, PROMELA, SPIN, services de sécurité, intrus, EFSM.

Abstract

An important part of cryptanalysis consists in seeking the intrinsic faults in the rules which define security protocols while supposing that cryptographic algorithms used in those protocols are unbreakable. These faults lead to attacks by illegitimate intruders.

In this study, we propose a model for automatic validation of combinations of several security protocols in order to demonstrate the conformity of their specifications that claim to cover some set of safety risks in a context of transaction between several participants.

Our model of validation is based on a formal specification language for distributed protocols PROMELA and on a tool for simulating concurrent protocols called SPIN.

One of the most critical parts in the validation model is the exhaustive description of the behavior of an intruder in the environment where the communication to be protected takes place. Considering that such a behavior is not deterministic in arbitrary contexts of confidence, the model of security protocol validation must be rendered as deterministic as possible. Therefore, we use EFSMs (*Extended Finite State Machines*) to describe the behavior of the intruders.

We demonstrate the functioning of our validation model by a case study featuring a combination of different security mechanisms. The set of mechanisms chosen includes : an electronic signature to assure data integrity and non-repudiation, a numeric certificate for the authentication of participants, and a symmetric cryptography algorithm to guarantee confidentiality.

Key words : Validation, security mechanism, security protocol, formal specification, SecAdvise, PROMELA, SPIN, security services, intruder, EFSM.

Table des matières

Table des figures

Liste des tableaux

Remerciements

Je tiens à remercier mes directeurs de recherche le professeur Peter Kropf et le professeur Gilbert Babin de m'avoir donné la chance d'accomplir ce travail sous leur direction, du temps qu'ils ont bien voulu me consacrer, des bonnes idées qu'ils m'ont données, de leur disponibilté et de la qualité de leur encadrement.

J'aimerais finalement remercier mes parents, Hammad Zemmouri et Radia Idrissi, ainsi que mes sœurs Maria et Salma pour leurs conseils et leur soutien moral et financier.

Chapitre 1

Introduction

1.1 Contexte du travail

Toute transaction commerciale repose sur la confiance mutuelle des intervenants en leur probité, dans la qualité des biens échangés et dans la fiabilité des systèmes de transfert de paiement ou de livraison des achats. Puisque les échanges associés au commerce électronique se déroulent la plupart du temps à distance, un climat de confiance est nécessaire à la conduite des affaires même si les participants ne se rencontrent pas ou s'ils utilisent des monnaies dématérialisées. Les fonctions de sécurisation du commerce électronique interviennent à trois niveaux : celui du réseau, celui des échanges financiers proprement dits et enfin celui de la marchandise elle-même.

Nous nous concentrons dans ce mémoire à l'étude de la sécurisation des échanges financiers qui consiste à identifier et authentifier les intervenants afin de leur accorder les services auxquels ils ont souscrit, confirmer l'intégrité des échanges et, si besoin, leur confidentialité et enfin retenir les preuves susceptibles de régler les litiges éventuels. Les mesures de protection peuvent néanmoins contrarier d'autres attentes des usagers comme celle de l'anonymat et de la non-traçabilité des transactions.

1.2 Résultats attendus

Le but de cette recherche est de compléter la conception de l'aviseur SecAdvise [22, 21] conçus pour choisir des solutions de sécurité optimales pour des entités voulant communiquer sûrement. Nous avons ajouté les notions de couverture de risque, de

contraintes, de coût et de validation de solution.

Le pourcentage de couverture de risque diffère d'un mécanisme de sécurité à un autre ce qui pousse à développer de nouvelles formules pour calculer un ensemble de mécanismes de sécurité permettant une couverture maximale des risques menaçant une transaction sous un ensemble de contraintes (économiques, politiques, etc.) avec un coût minimum. La validation d'une solution, sélectionnée par SecAdvise et prétendant couvrir un ensemble de risques, par une méthode de vérification automatique est un des point les plus importants dans la sélection.

Cette validation consiste à trouver toute faille intrinsèque dans toute solution de sécurisation. Elle met en jeu des automates représentant le fonctionnement de chaque participant d'une part et d'une autre part un automate représentant le comportement d'un intrus avec tout le pouvoir que peut posséder un intrus réel qui essaie d'intercepter l'échange dans un réseau ouvert.

1.3 Structure du mémoire

Dans le chapitre 2, nous présentons une vue d'ensemble des services de sécurité, des mécanismes et protocoles de sécurité disponibles avec la description de quelques algorithmes de cryptographie sur lesquels se base toute l'infrastructure de sécurité des transactions électroniques. Dans le chapitre 3, nous rappelons la conception de base de l'aviseur SecAdvise et nous décrivons les nouvelles notions à ajouter à cet aviseur pour renforcer la confiance de ses utilisateurs. Le chapitre 4 présente l'outil PROMELA/SPIN, un paquetage pour la spécification formelle des systèmes distribués et pour la simulation/validation de ces systèmes. Le chapitre 5 aborde l'étude de cas sur laquelle on a appliqué l'outil PROMELA/SPIN ainsi que les résultats de la vérification. Le chapitre 6 fournit une description approximative de la génération automatique du comportement d'un intrus. Dans le dernier chapitre nous concluons et identifions les travaux futurs pouvant découler de cette recherche.

Chapitre 2

État de l'art

L'objectif de ce chapitre est de cerner le concept de la sécurité dans le contexte du commerce électronique, et plus particulièrement des transactions électroniques, ainsi que les risques menaçant le bon déroulement de ces transactions. Nous parcourons les solutions cryptographiques liées à chaque service de sécurité, à savoir l'authentification, l'identification des participants, la confidentialité et l'intégrité des données. Ensuite, nous décrivons quelques mécanismes de sécurité, SSL, SET et PGP ainsi que les algorithmes de cryptographie qu'ils utilisent afin d'assurer les services de sécurité qu'ils prétendent couvrir.

2.1 Le commerce électronique

L'association française pour le commerce et les échanges électroniques l'AFCEE[1], définit le commerce électronique comme « ***l'ensemble de relations totalement dématérialisées que les agents électroniques ont les uns envers les autres*** ».
Cette définition a l'avantage de couvrir toute la panoplie des transactions électroniques qu'elles soient réalisées par l'Internet, les réseaux bancaires, les intranets, les réseaux sans fils ainsi que les portes-monnaies et les chèques électroniques.

Parmi les catégories des applications du commerce électronique on trouve :

1. le commerce inter-entreprises (B2B) : le client est une autre entreprise ou un autre service d'une autre entreprise ;

[1] http ://www.afcee.asso.fr

2. le commerce grand public (B2C) : il permet à un particulier d'effectuer une transaction électronique à distance via un réseau de télécommunication ;
3. le commerce de proximité qui inclut les deux types B2C et C2C et qui se caractérise par un face-à-face entre l'acheteur et le vendeur.

On trouve aussi d'autres catégories qui sont apparues dernièrement comme le G2B qui englobe l'ensemble des transactions commerciales en ligne qui sont spécifiquement réalisées entre un gouvernement et une entreprise, le G2C, etc.

Le développement du commerce électronique passe par un ensemble de solutions matérielles et logicielles capables de répondre aux attentes des utilisateurs. Une architecture de commerce électronique doit tenir compte de la sécurisation des transactions, des échanges d'information et des différentes conditions d'accès aux réseaux de télécommunications. Nous nous limitons dans ce chapitre à l'étude de la sécurité de l'aspect logiciel et plus précisément de la sécurité dans les échanges et transactions effectuées sur un réseau ouvert ou publique.

Parmi les solutions logicielles, on trouve de plus en plus les applications Web dont l'utilisation est devenue très courante au point qu'elle soit transparente à l'utilisateur. Des exemples d'utilisation des applications Web : le transfert d'argent via un site bancaire, l'achat de toute sorte de marchandise, des enchères d'appels d'offre publiques, ou encore la recherche de l'heure d'arrivée et de départ d'un vol sur un site d'une compagnie aérienne.

Les applications Web et les services Web (applications Web qui décrivent leur fonctionnement à d'autres applications Web afin de leur permettre de les utiliser) donnent naissance à une troisième génération de l'Internet. Sun et Microsoft avec respectivement leur stratégie Sun ONE (*Open Net Environement*) et .NET forment de plus en plus l'infrastructure de l'Internet.

La question qui s'impose d'avantage, est à savoir s'il y a une protection adéquate devant l'énorme vague de données qui migrent de plus en plus sur le Web. À ce jour, aucune technologie Web n'a démontré qu'elle est à l'abri de l'inévitable découverte de vulnérabilités affectant la sécurité et la confidentialité de ses propriétaires et ses utilisateurs.

2.2 Les applications Web

Une application Web est un logiciel client/serveur [4] qui interagit avec des utilisateurs ou d'autres systèmes en utilisant le protocole HTTP. Pour un utilisateur un client pourrait être un fureteur Web comme Internet Explorer ou Netscape Navigator. Pour une autre application, cela pourrait être un agent HTTP qui agit comme un explorateur automatique.

L'utilisateur visualise les pages Web et est capable d'interagir en envoyant des choix au système. Les actions qui s'en suivent varient de tâches simples comme chercher le chemin local d'un fichier ou d'un répertoire, à des tâches plus sophistiquées comme les applications qui assument des transactions en temps réel entre plusieurs tiers, incluant les applications du commerce électronique B2B et B2C, la gestion électronique de processus (*workflow*) ou les applications patrimoniales (*legacy application*).

Les applications modernes sont en majorité développées en Java (ou langages similaires) et distribuées sur plusieurs serveurs reliés à de multiples sources de données via des entités logiques. Une application Web se compose habituellement de trois couches logiques :

La couche présentation est responsable de la présentation finale des données à l'utilisateur du système. Elle comprend des serveurs Web comme Apache ou Internet Information Server et aussi les fureteurs Web comme Internet Explorer ou Netscape Navigator. Elle peut comprendre aussi des composantes pour la mise en page. Le serveur Web cherche les données et le fureteur les présente sous une forme lisible à l'utilisateur. Cela permet aussi à l'utilisateur d'interagir avec l'application en envoyant au serveur Web des paramètres.

La couche application est la partie la plus importante d'une application Web parce qu'elle se charge du côté opérationnel de l'application en traitant les entrées de l'utilisateur, en prenant des décisions, en faisant des requêtes de données et en les présentant à la couche présentation. Elle utilise des produits comme IBM WebSphere, WebLogic, JBOSS ou ZEND qui sont développés avec des technologies comme CGI (*Common Gateway Interface*), Java, .NET, PHP (*Hypertext*

Preprocessor), Sun ONE ou ColdFusion.

La couche de données est une banque de données pouvant être permanentes ou temporaires. Le format XML est de plus en plus utilisé dans la couche de données pour des raisons d'interopérabilité avec les autres systèmes.

2.3 Les services Web

2.3.1 Définition d'un service Web

Les services Web [5] sont des services applicatifs accessibles via des protocoles standardisés du Web par des entités distantes (applications ou utilisateurs). L'Internet est donc en train de passer de l'ère du client/serveur à celui de l'application à application. Deux domaines d'utilisation sont mis de l'avant par les acteurs du marché : la location d'application ou ASP (*Application Service Providing*) et les applications distribuées.

Le premier domaine permet par exemple à un particulier de consulter son agenda ou son portefeuille en ligne, de sauvegarder des documents sur un serveur distant, etc. C'est un modèle économique qui pour l'instant ne décolle pas. Quant au second, il permet aux entreprises de passer d'un modèle de commerce interne à un modèle externe. Au lieu de seulement automatiser leurs processus de commerce interne, elles construisent des sites Web qui intègrent à la fois leurs services Web mais aussi ceux de leurs partenaires. Cette collaboration est appelée « automatisation du commerce Internet ».

2.3.2 Protocoles des services web

Les protocoles les plus importants dans la technologie des services Web sont SOAP, WSDL et UDDI.

SOAP (*Simple Object Access Protocol*)[2] est un document de travail du W3C (*World-Wide Web Consortium*).

La spécification SOAP définit un modèle d'échange de messages qui repose sur trois concepts :

– les messages sont des documents XML,

[2]http ://www.w3.org/TR/soap12/

- ils voyagent d'un émetteur vers un récepteur,
- les récepteurs peuvent former une chaîne.

Les messages SOAP peuvent donc suivre le modèle client/serveur classique via HTTP ou être utilisés suivant le modèle d'échange de messages à sens unique. Dans les deux cas, on trouvera un émetteur (E) et un récepteur, appelé point final (PF). En chaînant ces messages, on obtient une suite d'émetteurs et de récepteurs SOAP appelés intermédiaires SOAP.

WSDL (*Web Service Description Language*)[3] est une norme dérivée de XML qui décrit l'interface d'utilisation d'un service Web (méthodes et propriétés des composants de l'application), ses liaisons de protocoles et ses détails de déploiement. La description de l'interface par un fichier WSDL est comparable à la partie publique d'une classe ou à un en-tête (*header*) d'un programme.

UDDI (Universal Description Discovery and Integration)[4] est un protocole d'annuaire permettant de trouver le service Web que l'on cherche (de façon manuelle ou automatique), mais aussi d'en annoncer la disponibilité. UDDI utilise SOAP comme couche de transport. Un annuaire UDDI (*UDDI business registry*) est constitué de pages blanches (nom de l'entreprise, adresse, contacts), jaunes (services classés par catégories industrielles) et vertes (informations d'implémentation des services Web proposés). Toute information saisie dans un annuaire et répliquée sur l'ensemble des annuaires. Concrètement, ces annuaires sont des fichiers XML hébergés par des entreprises appelées opérateurs UDDI ou nœuds d'opérateurs. UDDI recommande d'utiliser WSDL pour la description des services. Grâce à UDDI les entreprises peuvent enregistrer des données les concernant, des renseignements sur les services qu'elles offrent et des informations techniques sur le mode d'accès à ces services.

Qu'il s'agisse d'application Web ou de services Web, ces systèmes émergeants font face aux mêmes problèmes de sécurité que les applications du commerce électronique traditionnelles. La section qui suit décrit les risques de sécurité dans les transactions Web.

[3] http ://www/w3.org/TR/wsdl
[4] http ://www.uddi.org

2.4 Risques de sécurité

Dans la partie intitulée « Architecture de sécurité » de la norme multi-parties ISO 7498 [13], on trouve des définitions et des concepts de sécurité de base. Le présent chapitre ainsi que le suivant s'en inspirent largement.

Plusieurs types de menaces informatiques se rencontrent dans un réseau ouvert. Les recommandations X.509 [25] et X.800 [24] de l'UIT-T (*Union Internationale des Télécommunications*) signalent entre autres les risques de sécurité suivants :

- l'interception de l'identité d'un ou de plusieurs des intervenants par un tiers en vue d'un usage abusif ;
- l'usurpation d'identité ;
- la réexécution (ou le rejeu) intégrale ou partielle d'une communication légitime antérieure après enregistrement ;
- l'interception de données par un intrus ou un utilisateur non autorisé au moyen d'une observation clandestine des échanges pendant une communication ;
- la modification accidentelle ou délictueuse du contenu des échanges par remplacement, insertion, suppression ou réorganisation de données d'utilisateur au cours d'une communication ;
- la dénégation ou contestation d'un utilisateur d'avoir participé, partiellement ou entièrement, aux échanges d'une communication ;
- le déni de service et l'impossibilité d'accès à des ressources habituellement mises à la disposition des utilisateurs autorisés à la suite d'un empêchement, d'une interruption de la communication ou de délais importants imposés à des opérateurs critiques ;
- l'acheminement erroné d'un message prévu pour un usager vers un autre usager ;
- l'analyse de trafic et l'examen des paramètres relatifs à une communication entre utilisateurs (c'est-à-dire absence/présence, fréquence, sens, séquence, type, volume, etc.). Cette analyse peut être rendue plus difficile en intercalant un trafic inintelligible au sein des données utiles (remplissage de trafic) et en utilisant des données chiffrées ou aléatoires.

Pour couvrir les risques cités ci-haut, les solutions de sécurité doivent avoir comme

objectifs l'interdiction à un tiers non autorisé de lire ou de manipuler le contenu des messages échangés sans risque d'être détecté, l'entrave de truquage des pièces ou la génération de messages, la satisfaction des conditions légales en vigueur pour la validation des contrats et le règlement des litiges, l'assurance de l'accès aux services souscrits selon les contrats établis par les fournisseurs, et enfin l'assurance d'un niveau de service égal à tous les clients quelle que soit leur localité géographique. L'objectif des services de sécurité est de minimiser les conséquences de telles erreurs, sinon de les prévenir. La section qui suit définit les services de sécurité en réseau ouvert ainsi que les mécanismes de sécurité couvrant les risques de sécurité cités ci-dessus.

2.5 Services de sécurité

La sécurisation des échanges du commerce électronique consiste à utiliser des fonctions mathématiques pour brouiller le texte initial avant sa transmission. Ce texte devra être restitué sous sa forme initiale après réception par le destinataire authentique. La sécurisation comporte cinq services qui sont décrits ci-dessous.

2.5.1 La confidentialité des messages

La confidentialité garantit que les informations sont communiquées uniquement aux parties autorisées à les recevoir. La dissimulation est réalisée à l'aide d'algorithmes de chiffrement. On reconnaît deux types de chiffrement : la cryptographie symétrique et la cryptographie à clé asymétrique.

La cryptographie symétrique

La clé utilisée par l'expéditeur pour chiffrer un message secret est la même que celle qu'utilise le destinataire légitime pour le déchiffrement. L'échange de la clé entre les partenaires doit être effectué via d'autres canaux de communication sécurisés.

Il y a deux catégories d'algorithmes de chiffrement symétrique [10] :

- les algorithmes de chiffrement **en bloc** qui agissent par transformation de blocs de données de taille fixe, généralement 64 bits, en blocs chiffrés de même taille,
- les algorithmes de chiffrement **enfilé** qui convertissent le texte en clair bit par

bit en combinant la suite de bits en clair et la série de bits de la clé de chiffrement à l'aide d'un *ou exclusif*.

Parmi les algorithmes de chiffrement les plus utilisés dans le commerce électronique on trouve : DES, IDEA, RC2, RC4, RC5, SKIPJACK,Triple DES ou TDEA.

Le principal inconvénient des systèmes cryptographiques symétriques est que les deux parties doivent, d'une manière ou d'une autre posséder l'unique clé de chiffrement. Au sein d'une organisation fermée cette contrainte ne pose pas de problème, mais sur des réseaux ouverts, l'échange peut être intercepté. La cryptographie à clé publique permet de résoudre le problème d'échange de clé.

La cryptographie à clé publique

Elle fait intervenir une paire de clés pour chaque participant, l'une privée (K_S) et l'autre publique (K_P). Les clés sont choisies de sorte qu'il soit difficile de reconstituer la clé privée à partir de la clé publique. La clé publique est la clé de chiffrement et la clé privée est la clé de recouvrement.

L'algorithme standard pour le chiffrement à clé publique est l'algorithme RSA (du nom de ses inventeurs *Ronald Rivest*, *Adi Shamir* et *Leonard Adleman*).

2.5.2 L'intégrité des données

Le but du service d'intégrité est d'empêcher toute modification non autorisée des messages durant leur parcours entre l'expéditeur et le destinataire. On utilise une séquence de bits associée d'une manière univoque au document à protéger. Cette séquence de bits constitue l'**empreinte** du document.

Le destinataire recalcule la valeur de l'empreinte à partir du message reçu et compare le résultat obtenu avec la valeur qui lui a été envoyée. Toute différence indique que le message n'a pas été préservé.

L'empreinte peut être construite en appliquant une **fonction de hachage**. Une fonction de hachage convertit une chaîne de caractères de longueur quelconque en une de taille fixe, généralement plus petite que la chaîne initiale, appelée **condensât**. C'est une fonction relativement simple à calculer dans un sens, mais considérablement plus difficile à calculer dans le sens inverse. Elle doit satisfaire les propriétés suivantes :

1. absence de collision : la probabilité d'obtenir le même condensât à partir de deux textes différents est minime, cette probabilité est inversement proportionnelle à la taille du condensât
2. impossibilité d'inversion : étant donné l'empreinte h d'un message m, il est difficile de calculer le message m tel que $H(m) = h$ avec h la fonction de hachage ;
3. une grande dispersion : une modification du texte initial, même minime, devra être répercutée sur le condensât.

Les fonctions de hachage les plus utilisées dans les applications du commerce électronique sont : AR/DFP, DSMR, MCCP, MD4, MD5, NVB701, NVBAK, RIPEMD, RIPEMD-128, RIPEMD-160 et SHA.

Vérification de l'intégrité avec la cryptographie à clé publique

Il y a trois types de signature par algorithme à clé publique :

Griffe un expéditeur chiffre son message avec une clé privée réservée pour l'opération de signature. Il rattache le résultat produit au message avant de l'envoyer. Toute personne ayant connaissance de la clé publique correspondante est en mesure de déchiffrer la griffe et de vérifier qu'elle correspond bien au message réceptionné.

Sceau on emploie un abrégé du message initial avant le chiffrement, pour cela une fonction de hachage à sens unique est utilisée pour produire l'empreinte du message qui sera ensuite chiffrée à l'aide de la clé privée de l'expéditeur.

Estampillage (ou signature en aveugle) une procédure spéciale pour faire signer un message par un notaire à l'aide de l'algorithme RSA de cryptographie à clé publique, sans lui en révéler le contenu. Une des utilisations possibles de cette technique est l'horodatage d'un paiement numérique. Les différents protocoles de paiement par monnaie numérique exploitent la signature en aveugle pour satisfaire les conditions d'anonymat.

Les algorithmes à clé publique de signature numérique les plus utilisés pour sceller les messages sont : DSA, ElGamal et RSA. L'emploi de DSA est imposé par le gou-

vernement fédéral des États-Unis dans tout système de chiffrement à clé publique employé par ses agences et départements.

Vérification de l'intégrité par la cryptographie à clé symétrique

Le **paraphe** ou code d'authentification de message (MAC : *Message Authentication Code*) est le produit d'une fonction de hachage à sens unique qui dépend d'une clé secrète. Le paraphe assure l'intégrité du contenu du message et l'authentification de l'expéditeur (vu que le MAC dépend d'une clé secrète que seuls les deux parties possèdent).

Pour construire un paraphe, on chiffre le condensât produit par une fonction de hachage à sens unique avec un algorithme de chiffrement de blocs. Ce paraphe est ensuite accolé au message en clair et le tout est expédié au destinataire. Celui-ci recalcule le condensât en appliquant la même fonction de hachage au message reçu et compare le résultat obtenu avec le paraphe déchiffré. L'égalité des deux résultats confirme l'intégrité des données.

2.5.3 L'identification des participants

Il s'agit de vérifier la relation entre des caractéristiques individuelles (des mots de passe ou des clés de chiffrement) et les individus, afin de contrôler l'accès aux ressources du réseau ou aux services offerts. Une entité peut posséder plusieurs identificateurs distincts. Elle est distincte de l'authentification qui est la confirmation que l'identificateur correspond bien à l'utilisateur déclaré.

L'authentification et l'identification d'une entité communicante ont lieu simultanément si cette entité propose en privé au vérificateur un secret partagé uniquement entre eux. La signature numérique est le moyen usuel d'identification car elle associe un utilisateur à un secret partagé entre les deux entités communicantes.

2.5.4 L'authentification des participants

L'authentification d'un participant est la preuve que l'identité que revendique une entité lui appartient. L'authentification est nécessaire pour assurer la non-répudiation.

Elle a pour objectif de réduire sinon les risques d'usurpation d'identité en vue de poursuivre des opérations non autorisées.

Lorsque les participants utilisent un **algorithme de chiffrement symétrique**, ils sont les seuls à partager une clé secrète. Par conséquent, l'utilisation d'un tel algorithme garantit à la fois **la confidentialité** des messages, **l'identification** correcte des interlocuteurs et leur **authentification**.

En revanche, lorsque les participants utilisent des **algorithmes à clé publique**, l'authentification consiste à demander à l'utilisateur de faire la preuve qu'il détient la clé privée correspondant à la clé publique qui lui est attribuée. **Une autorité de certification** assure l'association de la clé publique (et donc de la clé privée correspondante) à l'identité reconnue en délivrant un **certificat**.

Un annuaire est une base de données d'authentification, qui renferme les données relatives aux clés de chiffrement privées telles que leur valeur, la durée de leur validité, l'identité des possesseurs. Tout utilisateur est susceptible d'interroger cette base de données pour récupérer la clé publique du correspondant ou pour vérifier la validité du certificat que ce dernier est amené à présenter.

Le certificat garantit la correspondance entre une clé publique donnée et l'entité dont le nom distinctif unique est contenue dans le certificat. Ce certificat est scellé par la clé privée de l'autorité de certification. Lorsque le détenteur d'un certificat signe ses documents avec la clé privée de l'algorithme de chiffrement à clé publique, ses interlocuteurs peuvent vérifier la validité de sa signature à l'aide de la clé publique correspondante contenue dans le certificat. De même, pour envoyer un message confidentiel à une entité certifiée, il suffit d'interroger l'annuaire pour récupérer la clé publique de cette entité afin de chiffrer les messages que seul le détenteur de la clé privée associée est capable de déchiffrer.

2.5.5 La non-répudiation

La non-répudiation est un service qui permet d'éviter qu'une personne ayant accompli une action puisse la récuser plus tard, en partie ou en totalité. Il s'agit de fournir la preuve de l'intégrité des données et de leur origine de manière irréfutable et vérifiable par un tiers, par exemple la non-répudiation de l'envoi du message par

l'expéditeur ou sa réception par le destinataire. Ce service est aussi appelé authentification de l'origine des données.

Selon la recommandation X.813 de l'UIT-T, une solution pour la non-répudiation dans un système ouvert doit assurer les opérations suivantes :

- la génération des preuves,
- l'enregistrement des preuves,
- la vérification des preuves engendrées et
- le rappel et la vérification des preuves.

On dénombre deux types de services de non répudiation :

la non-répudiation de l'origine qui protége un destinataire confronté à un expéditeur niant avoir envoyé le message et

la non-répudiation de la réception qui joue le rôle inverse du précédent, à savoir démontrer que le destinataire a bien reçu le message que l'expéditeur lui a envoyé.

2.5.6 Services de sécurité et le modèle OSI

Les services de sécurisation peuvent se faire sur une ou plusieurs couches du modèle OSI. Le choix de la couche dépend des critères suivants :

- l'intervention a lieu au niveau de **la couche physique** ou de **la couche de liaison** si la protection de tous les flux doit être assurée de la même manière. La confidentialité est le seul service prévu au niveau de ces deux couches. L'inconvénient de la protection à ce niveau est qu'une attaque réussie déstabilise tout l'édifice de sécurité, car la même clé est utilisée pour sécuriser toutes les transmissions ;
- le chiffrement sera effectué au niveau de **la couche réseau** pour une protection sélective mais globale de toutes les communications associées à un sous-réseau particulier d'un point à un autre point ;
- pour assurer une protection avec restauration en cas de panne ou si **la couche réseau** n'est pas suffisamment fiable, c'est **la couche transport** qui fournit de bout-en-bout les services de sécurité suivants : l'authentification par mot de passe ou authentification simple, l'authentification par signatures ou certificats,

dite authentification poussée, le contrôle d'accès, la confidentialité et l'intégrité ;
- si un niveau plus fin de protection est recherché ou si le service de non-répudiation doit être assuré, le chiffrement se fait au niveau de **la couche application**. C'est à ce niveau qu'agissent la majorité des protocoles de sécurisation du commerce électronique.

2.5.7 Lézardes des services de sécurisation

Si le rôle du chiffrement est de brouiller les messages, la cryptanalyse a pour objectif de détecter les failles dans les algorithmes cryptographiques afin de déchiffrer les messages encryptés. Les attaques cryptographiques les plus connues sont des types suivants :

- attaque brute au cours de laquelle l'intrus essaie toutes les clés de chiffrage possibles, jusqu'à obtention de celle qui donnera le texte en clair ;
- attaque sur le texte chiffré à partir d'hypothèses sur le texte en clair dont on connaît l'agencement, par exemple la présence d'un en-tête au format connu ;
- attaque à partir du texte intégral en clair afin de découvrir la clé de chiffrement par comparaison avec les textes chiffrés correspondants ;
- attaque par rejeu d'anciens messages légitimes afin de déjouer les mécanismes de défense et court-circuiter le chiffrement ;
- attaque par interception de la clé secrète ou l'injection de faux messages qui seraient considérés comme légitimes par les deux parties ;
- attaque par mesure de la durée des chiffrements, des émissions électromagnétiques, etc., afin de déduire la complexité des opérations et, par conséquent, leur forme.

Certaines mesures sont recommandées pour parer aux attaques [1]. Citons à titre d'exemple :

- l'indication explicite de l'identité des participants, si elle est essentielle pour l'interprétation de la sémantique du message ;
- le choix d'une clé suffisamment longue pour décourager les attaques brutes ;
- l'ajout d'éléments aléatoires, par exemple un horodatage, pour rendre les attaques par rejeu plus difficiles.

2.5.8 Comparaison entre la cryptographie symétrique et la cryptographie à clé publique

Les systèmes basés sur les algorithmes à clé symétrique posent le problème de la distribution confidentielle des clés. Cela se traduit impérativement par un canal de distribution sécurisé et préétabli entre participants. En outre, chaque entité doit posséder autant de clés que d'interlocuteurs avec lesquels elle rentrera en contact. On conçoit que la difficulté de gestion de clés symétriques augmente quadratiquement avec le nombre de participants.

Ces difficultés sont évitées dans les algorithmes à clé publique, puisque chaque entité possède une seule paire de clés privée et publique. Cependant, les calculs qu'exigent les procédés à clé publique sont beaucoup plus lourds que ceux employés pour la cryptographie symétrique. Même si la compression des données avant le chiffrement avec la clé publique arrive souvent à accélérer les calculs, on réserve le chiffrement à clé publique pour assurer la confidentialité des messages courts.

Par conséquent, la cryptographie à clé publique est surtout considérée comme un complément à la cryptographie à clé symétrique. Elle permet d'assurer la distribution de la clé secrète en toute sécurité. Une nouvelle clé secrète est distribuée au début de chaque nouvelle session et dans les cas extrêmes, avant chaque échange, c'est le cas du protocole SET (*Secure Electronic Transaction*) que nous présentons dans la section qui suit.

2.6 Mécanismes de sécurité

2.6.1 Le protocole SSL

Le protocole SSL (*Secure Socket Layer*) [8] est employé pour sécuriser tous les échanges entre un client et un serveur de manière transparente.

Les services de sécurisation de SSL

L'authentification L'authentification utilise un certificat conforme à la recommandation X.509 version 3 de l'UIT-T [25]. Elle a lieu à l'établissement de la session et avant la première transmission de données. Ce service est obligatoire pour le serveur

dans la version 3.0 de SSL [8]. Cependant, le serveur peut exiger du client qu'il s'authentifie et peut même lui refuser l'établissement de la connexion en l'absence de certificat. L'échange de clé durant cette phase peut se faire avec l'algorithme RSA (la taille de la clé utilisée pour l'échange de clé est limitée à 512 bits), Diffie-Hellman ou Fortezza.

La confidentialité La confidentialité des messages s'appuie sur des algorithmes de chiffrement symétrique avec une clé de longueur de 128 bits ou de 40 bits. Le même algorithme cryptographique est utilisé par les deux parties mais chacune se sert de sa propre clé secrète, qu'elle partage avec l'autre partie : *client_write_key* et *server_write_key*. Les algorithmes que l'on peut exploiter sont : DES, DES40, 3DES, RC2, RC4-128, RC4-40 (dans le cas d'une exportation depuis les États-Unis), IDEA et SKIPJACK.

L'intégrité Elle est assurée par l'application d'une fonction de hachage, SHA ou MD5.

Les sous-protocoles de SSL

Le protocole SSL se compose de quatre sous protocoles :

Handshake est chargé de l'authentification, de la négociation des algorithmes de chiffrement et de hachage et de l'échange d'un secret, le $PreMasterSecret$. Les algorithmes négociés sont listés dans le tableau 2.1 ;

Record met en œuvre les paramètres de sécurité négociés pour protéger les données d'application ainsi que les messages en provenance des protocoles *Handshake*, *ChangeCipherSpec* (CCS) et *Alert* ;

ChangeCipherSpec a pour fonction de signaler au protocole *Record* toute modification des paramètres de sécurité ;

Alert est chargé de signaler les erreurs rencontrées pendant la vérification des messages ainsi que toute incompatibilité qui pourrait survenir pendant le protocole *Handshake*.

Fonction	*Algorithme*
échange de clés	RSA, Fortezza, Diffie-Hellman
Chiffrement symétrique à la volée	RC4 avec clés de 128 bits ou de 40 bits
Chiffrement symétrique en blocs	DES, DES40, 3DES, RC2, IDEA, Fortezza
Hachage	MD5, SHA

TAB. 2.1 – Algorithmes négociés par le protocole *Handshake*

Déroulement des échanges SSL

Les échanges définis par le protocole SSL se déroulent en deux temps :

1. durant la phase préliminaire ont lieu l'identification des parties, la négociation des attributs cryptographiques, la génération et le partage des clés ;
2. durant les échanges de données, la sécurisation opère à partir des algorithmes et des paramètres secrets négociés durant la phase préliminaire.

Chaque fois qu'un client se connecte à un serveur, il déclenche une session SSL. Si le client se connecte à un autre serveur, il engage une nouvelle session sans interrompre la session en cours.
SSL suggère de limiter la durée d'une session à 24 heures au maximum. Une session peut contenir plusieurs connexions contrôlées par l'application. La section 5.2.2 du chapitre 5 explique en détails le format de ses messages ainsi que les étapes de leurs échanges entre un client et un serveur.

Variables d'état d'une session SSL Les paramètres qui définissent une nouvelle session SSL sont :

- l'identificateur de session (*session ID*), une séquence arbitraire de 32 octets choisie par le serveur pour identifier une session active ou pouvant être réactivée ;
- le certificat du pair ;
- l'algorithme de compression ;
- la suite de chiffrement (*cipher_spec*) ;
- le *MasterSecret*, un secret de 48 octets partagé entre le client et le serveur, à partir duquel on génère les autres secrets. Ce paramètre est valable pour toute la session ;
- le drapeau (*is_resumable*), il signale s'il est permis d'ouvrir de nouvelles connex-

ions sur la session en question.

Une suite de chiffrement est définie par les cinq éléments suivants :

- la catégorie du chiffrement utilisé : enfilé (en continu), à la volée (en bloc) ;
- l'algorithme de chiffrement utilisé ;
- l'algorithme de hachage utilisé ;
- la taille du condensât ;
- la variable binaire signalant la permission d'exporter l'algorithme de chiffrement conformément à la loi étasunienne sur l'exportation de logiciels cryptographiques.

Il existe des restrictions quant aux combinaisons entre les algorithmes dans une suite de chiffrement négociée par un client et un serveur SSL. Par exemple, SSL n'accepte pas une suite de chiffrement où l'algorithme d'échange de clés est Diffie-Hellman, l'algorithme du chiffrement symétrique est RC4 et l'algorithme de Hachage est MD5. Le tableau 2.2 liste les combinaisons d'algorithmes pouvant être négociées par un client et un serveur SSL.

L'algorithme Diffie-Hellman est un algorithme d'échange de clés pour les algorithmes à clé publique. Il exploite la difficulté de calculer des logarithmes discrets sur un corps fini par rapport au calcul d'exponentielles sur le même corps. Le protocole SSL peut employer l'algorithme Diffie-Hellman éphémère, selon lequel les paramètres Diffie-Hellman sont signés à l'aide des algorithmes RSA ou DSS pour garantir leur intégrité. Les clés publiques des algorithmes utilisés pour la signature sont contenues dans des certificats signés par l'autorité certifiante.

Variables d'état d'une connexion SSL Les paramètres qui définissent l'état d'une connexion pendant une session sont ceux qui seront rafraîchis lors de l'établissement d'une nouvelle connexion. Ces paramètres sont :

- deux nombres aléatoires (*server_random* et *client_random*) de 32 octets. Les clés secrètes sont dérivées de ces nombres aléatoires ;
- deux clés secrètes, *server_MAC_write_secret* et *client_MAC_write_secret*, employées par les fonctions de hachage pour calculer les codes d'authentification (ou MAC) ;
- deux clés pour le chiffrement symétrique des données, *server_write_key* et

échange de clés	*Chiffrement symétrique*	*Hachage*	*Signature*
RSA	Sans chiffrement	MD5 ou SHA	
	RC4-40	MD5	
	RC4-128	MD5 ou SHA	
	RC2 CBC40	MD5	
	IDEA CBC	SHA	
	DES40 CBC	SHA	
	DES CBC	SHA	
	3DES EDE CBC	SHA	
Diffie-Hellman	DES40 CBC	SHA	DSS ou RSA
	DES CBC	SHA	DSS ou RSA
	3DES EDE CBC	SHA	DSS ou RSA
Diffie-Hellman éphémère	DES40 CBC	SHA	DSS ou RSA
	DES CBC	SHA	DSS ou RSA
	3DES EDE CBC	SHA	DSS ou RSA

TAB. 2.2 – Suites de chiffrement reconnues par SSL

client_write_key ;

- deux vecteurs d'initialisation pour le chiffrement symétrique ;
- deux numéros de séquence, chacun codé sur 8 octets.

2.6.2 Le protocole SET

SET (*Secure Electronic Transaction*) [26] est un protocole de sécurisation des transactions par cartes bancaires effectuées sur des réseaux ouverts, comme l'Internet.

Son but est de développer l'usage des cartes bancaires pour les paiements en ligne et d'éviter l'éclatement du marché entre une multitude de protocoles incompatibles entre eux. SET opère au niveau de l'application indépendamment de la couche de transport, ce qui le distingue de SSL.

En pratique, il est envisagé de l'utiliser pour sécuriser les transports conformes au protocole TCP. SET porte uniquement sur l'acte de paiement, excluant ainsi la recherche et la sélection des produits.

Dans une transaction SET, le porteur de la carte effectue son paiement en présentant un certificat que lui a délivré une autorité de certification. Ce certificat permet d'authentifier le porteur à l'aide de la cryptographie à clé publique.

Architecture de SET

Les architectes de SET ont eu pour principe fondamental de sécuriser les transactions par carte bancaire sur l'Internet sans modifier les circuits bancaires d'autorisation et de télécollecte existants.
Les principaux acteurs de SET sont :

- le détenteur de la carte bancaire (le porteur) ; il possède une carte, conforme aux spécifications SET, émise par un institut d'émission, une banque affiliée à VISA ou MasterCard ;
- le serveur du marchand ;
- la passerelle de paiement ;
- l'autorité de certification ;
- l'institut émetteur de la carte bancaire du porteur ;
- l'institut acquéreur, qui est souvent la banque du marchand.

Le porteur, le marchand, l'autorité de certification et la passerelle de paiement sont reliés par le réseau Internet. Le client n'établit pas de connexion directe avec la passerelle de paiement, mais utilise un tunnel passant par le serveur du marchand (technique de *tunneling*).

Chaque participant doit d'abord obtenir un certificat auprès d'une autorité de certification agréée selon les spécifications de SET. Ces certificats sont ensuite inclus dans les messages échangés entre le détenteur de la carte, le marchand et la passerelle de paiement. Les instituts d'émission et d'acquisition sont reliés par un réseau bancaire fermé et sécurisé. La passerelle de paiement fait le pont entre les deux réseaux, l'un ouvert et l'autre fermé, ce qui permet de protéger l'accès au réseau bancaire. Elle doit donc posséder deux interfaces, l'une pour le protocole SET (côté Internet), l'autre pour un protocole propriétaire (côté réseau bancaire). SET sécurise les échanges entre le client et le marchand ainsi qu'entre le marchand et la passerelle de paiement.

Le protocole SET est un protocole orienté transaction et fonctionne selon le mode requête/réponse ; en d'autre termes, les messages constituent des paires.

Les services de sécurité fournis par SET

Les transactions de SET fournissent les services suivants :

- inscription des porteurs et des marchands auprès de l'autorité de certification ;
- octroi des certificats aux porteurs et aux marchands ;
- authentification des participants ;
- confidentialité des transactions ;
- intégrité des transactions d'achat ;
- autorisation du paiement ;
- capture (collecte) du paiement pour initier la demande de compensation financière au profit du marchand.

SET utilise les techniques de cryptographie à clé publique afin de garantir à la fois :

la confidentialité des messages, elle est assurée à l'aide d'algorithmes de chiffrement. Cependant la clé secrète est elle-même distribuée à l'aide d'algorithmes de cryptographie à clé publique.

l'intégrité des données échangées entre le client, le marchand et la banque acquéreure ; SET utilise le sceau de l'expéditeur pour assurer l'intégrité du message (rappelant que le sceau est la signature numérique d'un message chiffré obtenu en chiffrant le condensât du texte en question à l'aide de la clé privée du signataire). Quiconque dispose de la clé publique de l'émetteur, pourra vérifier l'intégrité du message en comparant le condensât obtenu en déchiffrant le sceau au condensât recalculé. Le sceau garantit à la fois l'identité de l'émetteur et l'intégrité des données.

l'authentification et l'identification des intervenants dans une transaction SET correspondent à une relation préétablie entre une clé de chiffrement et une entité. Ainsi, chaque entité joint à son message, chiffré ou non, une signature numérique qu'elle seule peut générer mais qui peut être vérifiée par les entités paires.

2.6.3 PGP

PGP (*Pretty Good Privacy*) [9] est un ensemble de programmes qui assurent dans un échange de messages confidentialité, authentification, signature numérique et compression en s'appuyant en grande partie sur les algorithmes RSA, IDEA et MD5. Il est décrit dans le document RFC 1991 [12] de l'IETF.

Sa force consiste en sa combinaison des meilleures fonctionnalités de la cryptographie à clé publique ; facilité d'utilisation, résolution du problème de distribution de clé et de la transmission de données, et de la cryptographie à clé secrète qui est plus rapide que celle à clé publique.

PGP utilise des algorithmes de cryptographie pour assurer confidentialité, authentification et non répudiation. La figure 2.1 résume les étapes par lesquelles passe un message avant son envoi par PGP et qui sont :

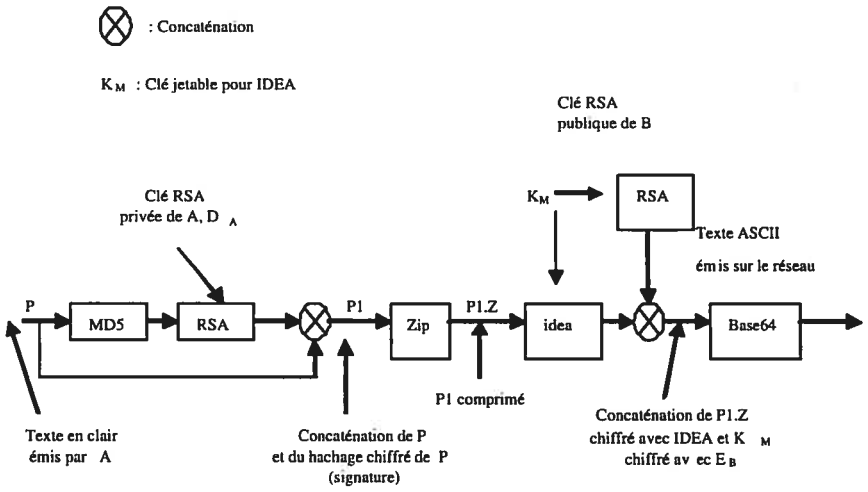

FIG. 2.1 – Fonctionnement de PGP

- un échange de clé publique RSA avec hachage MD5 ;
- une compression à l'aide du programme standard ZIP, ce qui réduit la taille du fichier et les redondances avant le chiffrement. La diminution de la taille du fichier augmente la vitesse de traitement et de transmission tandis que la réduction des redondances rend la cryptanalyse plus difficile ;
- le chiffrement du message à l'aide d'IDEA ;
- le chiffrement de la clé secrète de l'utilisateur obtenue à partir de l'empreinte d'une phrase-clé au lieu d'un mot de passe ;
- la segmentation du message.

2.6.4 Le protocole UDDI

Toute altération des données concernant une entreprise sera susceptible de nuire à son image, de nuire à la confiance portée par ses partenaires qui utilisent ou projettent d'utiliser ses services, et de nuire à la confiance en l'annuaire lui-même.

Un premier principe de sécurité de UDDI 2.0 est que toute information publiée sur un annuaire ne peut être modifiée que sur ce même annuaire et non sur un autre.
Un deuxième principe est que chaque appel aux fonctions de l'interface de programmation (API) de publication (ajout, modification, suppression) de UDDI nécessite une authentification à l'annuaire par le biais d'un jeton d'authentification. Pour ce faire, l'API de publication propose une fonction qui prend en paramètre une identité et un justificatif (obtenus lors de la création du compte sur l'interface Web du nœud d'opérateur) et retourne ce jeton.

Dans le cadre d'une sécurité accrue, il est aussi possible de s'authentifier directement (sans demande de jeton) via un certificat qui transmettrait directement le jeton (la valeur du certificat). Cette authentification permet alors de s'assurer que seul l'éditeur d'une entrée dans l'annuaire puisse la modifier ou la supprimer.

Par ailleurs, les informations sensibles qui nécessitent intégrité et confidentialité, telles que celles transmises lors de la création de compte ou lors de la gestion des entrées de l'annuaire, seront envoyées via SSL à travers HTTPS.

Pour s'assurer que les services UDDI fonctionnent correctement et de façon cohérente, la version 2 de UDDI a formalisé des spécifications destinées aux opérateurs UDDI. Elle recommande la mise en place de politiques pour trois aspects, à savoir l'accès non autorisé, la divulgation d'informations et le refus de services. Elle indique également quelles sont les informations devant être saisies en vue d'un audit (identité, heure, fonction de l'API, etc.) et les autorités de certification agréées pour les nœuds d'opérateurs.

2.6.5 Évaluations

SSL Le protocole SSL se trouve sur la plupart des plates-formes sécurisées de la Toile aussi bien du côté serveur que du côté client. Il offre les services d'authentification, de

confidentialité et d'intégrité des données dans les applications point à point. Son avantage par rapport à d'autres protocoles de sécurité est sa transparence par rapport aux applications sur TCP. Le protocole SSL a dépassé les applications transactionnelles comme le prouve son adoption par le consortium WAP (*Wirless Application Protocol*) pour les communications radiophoniques.

L'architecture modulaire de SSL permet de faire évoluer certaines parties du protocole sans remettre en cause l'ensemble de l'édifice. Il est donc possible d'introduire de nouveaux algorithmes plus éprouvés et de tenir compte des particularités réglementaires nationales.

L'utilisation de clés de 40 bits rend SSL vulnérable aux attaques par force brute. Cette contrainte a été imposée par les lois étasuniennes sur l'exportation. Une autre faiblesse, d'ordre structurel, provient du fait que les paramètres de chiffrement ne sont pas obligatoirement modifiés pendant une session. Le risque de casser les clés augmente avec la longueur de la session.

Enfin, SSL est inadapté aux applications du commerce électronique qui mettent en relation plusieurs acteurs, notamment le client, le marchand et une passerelle avec les réseaux bancaires. Pour tenir compte des relations d'intermédiation commerciales, il faut associer d'autres protocoles à SSL, ce qui risque d'alourdir l'architecture des systèmes de communication.

SET SET veut devenir le standard pour la sécurisation des paiements par carte bancaire sur l'Internet. Les caractéristiques essentielles de SET sont les suivantes :

- Le commerçant conserve les renseignements d'achat que signe le client au moyen de sa clé privée. Il possède aussi la réponse (grâce au message *AuthRes*) de la passerelle signée avec la clé privée de celle-ci. Éventuellement, le marchand aura aussi connaissance du certificat du porteur de carte avec la clé publique qui l'accompagne. Cependant, il ne possède pas les coordonnées de la carte bancaire du client.
- Le porteur de la carte possède la réponse à son ordre d'achat. Cette réponse est signée par la clé privée de signature du commerçant. Il possède aussi le certificat de signature du marchand, mais aussi son certificat de chiffrement.

- La passerelle de paiement a connaissance des caractéristiques financières de la transaction entre le commerçant et le porteur de carte sans connaître les articles vendus.
- À chaque transaction est attribué un numéro unique chiffré, ce qui protège contre les attaques par rejeu de transaction.

Néanmoins, les moyens de sécurisation sont assez compliqués, ce qui se traduit par une surcharge en calcul et des temps de réponse assez longs. SET a donc été sévèrement critiqué par la communauté cryptographique. La lourdeur de son fonctionnement exclut son utilisation pour les paiements de faibles montants. D'autres facteurs ont tendances à freiner l'usage du protocole SET, entre autres :

- les aspects juridiques, surtout lorsque la législation limite l'utilisation du chiffrement ;
- les secrets du porteur de la carte se trouvent stockés sur son disque dur, ce qui augmente les risques. Le protocole C-SET (*Cipher-Secured Electronic Transaction*), qui est une adaptation de SET aux cartes à puce, évite ce problème.

PGP *Pretty Good Privacy* est réputé offrir le système de sécurité commercial le plus proche du grade militaire. Quoique l'IETF (*Internet Engineering Task Force*) ait travaillé sur PGP, elle hésite à le sanctionner comme norme puisqu'il incorpore des algorithmes protégés par des brevets, notamment IDEA et RSA. Les activités en cours dans l'IETF tentent de réutiliser le cadre que définit PGP mais en employant des protocoles qui ne présentent pas ces inconvénients.

2.7 Algorithmes de cryptographie

2.7.1 L'algorithme de hachage MD5

Dans les applications à sécuriser, l'empreinte du message est calculée à l'aide d'une fonction à sens unique, une fonction relativement simple à calculer dans un sens mais considérablement plus difficile dans le sens inverse. La fonction choisie doit satisfaire les propriétés suivantes :

absence de collision, la probabilité pour obtenir le même condensât à partir de

deux textes différents est presque nulle. Pour que la probabilité de collision soit extrêmement réduite, la taille L du condensât (appelée aussi empreinte) doit être suffisamment grande. Par exemple la longueur du condensât de l'algorithme de hachage MD5 [18] utilisé par PGP est de 128 bits ;

impossibilité d'inversion ;

grande dispersion, une petite différence entre deux messages crée un grand écart entre leur condensât.

Ainsi toute modification du texte initial, même minime, devra être répercutée, en moyenne sur la moitié des bits du condensât.

Un message M est divisé en n blocs de longueur B bits chacun, les blocs de MD5 sont de 512 bits. On ajoute, si besoin des bits de remplissage à la fin du message selon un schéma précis afin que la taille de chaque bloc atteigne les B bits nécessaires. Les opérations de hachage cryptographique sont décrites à l'aide d'une fonction de compression f qui opère sur des blocs de données de longueurs B bits selon la relation récursive suivante :

$h_i = f(h_{i-1}, m_i),\ i = 1, \ldots, n$

où h_0 est le vecteur contenant la valeur initiale de L bits et

$m = m_1, m_2, \ldots, m_n$ est le message décomposé en n vecteurs de B bits chacun.

2.7.2 L'algorithme RSA

L'algorithme à clé publique RSA [18, 19] est fondé sur certains principes de la théorie des nombres. On peut résumer le fonctionnement du RSA en trois étapes principales :

1. Calcul des paramètres suivants :
   - deux nombres premiers, p et $q > 10^{100}$,
   - les produits $n = p.q$ et $z = (p-1).(q-1)$,
   - un nombre d premier avec z et
   - un nombre e tel que $e.d = 1\ (mod\ z)$.
2. Découpage du texte en clair (considéré comme une suite de bits) en une suite de blocs de telle sorte que chaque texte en clair, M, soit un nombre tel que

$0 \leq M < n$.

Ceci peut être fait en regroupant simplement les bits du texte en clair par bloc de k bits, où k est le plus grand nombre entier tel que $2^k < n$.

3. Pour chiffrer un message M, on calcule $C = M^e \ (mod\ n)$ et pour déchiffrer C, on calcule $M = C^d \ (mod\ n)$.

On peut démontrer que pour tout message M dans l'intervalle $[0, n[$, les deux fonctions de chiffrement et de déchiffrement sont bien inverses l'une de l'autre.
La clé publique de cet algorithme est donc le **couple** (e, n) et **la clé secrète**, le **couple** (d, n).

RSA est utilisé pour **distribuer des clés de session** utilisées avec DES, IDEA ou autres clés qui sont d'une longueur raisonnable. RSA est en effet trop lent pour qu'on puisse raisonnablement l'utiliser pour chiffrer de grandes quantités de données.

2.7.3 L'algorithme à clé symétrique IDEA

IDEA (*International Data Encyption Algorithm*) [17] utilise une clé de 128 bits et résiste donc assez bien à toute recherche exhaustive de clé et à toute autre attaque par collision.

La structure de base de l'algorithme consiste à embrouiller des blocs en clair en entrée par une suite d'itérations paramétrées qui produisent en sortie des blocs chiffrés de 64 bits. À chaque itération, chaque bit de sortie dépend de tous les bits d'entrée, cela rend le bruillage des données plus puissant.

La figure 2.2, décrit en détail une itération. On y effectue trois types d'opération sur des entiers non signés de 16 bits. Ces opérations sont des *ou exclusifs*, des *additions modulo 216* et des *multiplications modulo (216 + 1)*. Ces opérations ont la propriété qu'aucune paire n'obéit à une loi associative ou distributive, ce qui complique la cryptanalyse. La clé de 128 bits est utilisée pour engendrer des sous-clés de 16 bits, six pour chacune des huit itérations et deux pour la transformation finale. Le déchiffrement utilise le même algorithme que le chiffrement mais avec des sous clés différentes. Il y a eu des implémentations d'IDEA matérielles et logicielles.

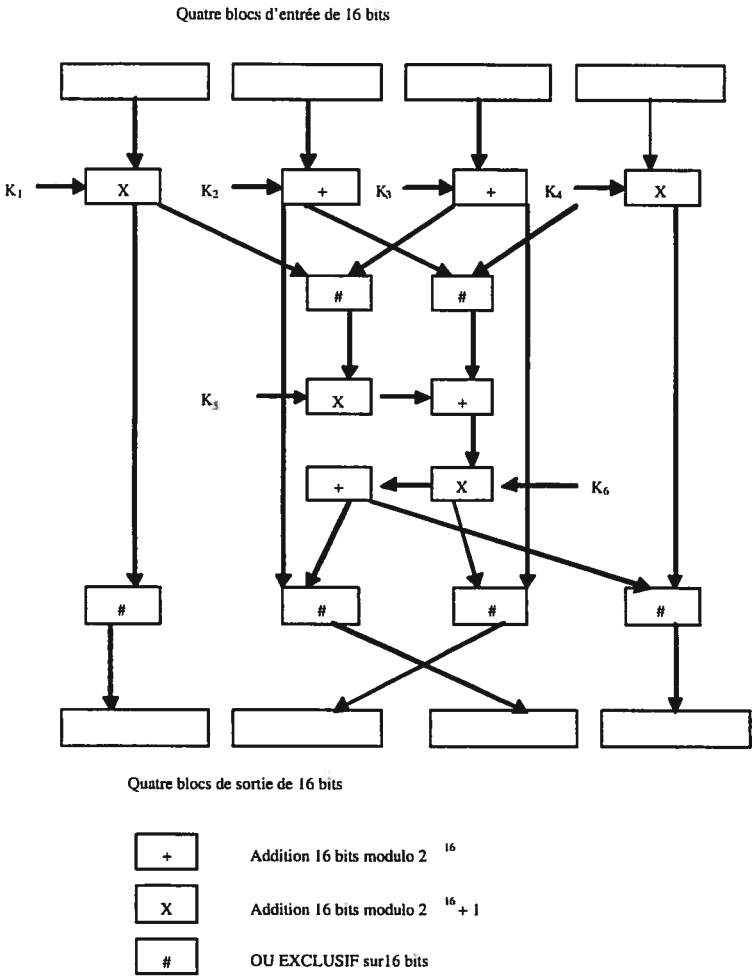

FIG. 2.2 – Détails une itération IDEA

2.7.4 Certificat numérique X.509

Dans un environnement de sécurité utilisant des clés publiques, un certificat numérique assure que la clé publique avec laquelle le message est crypté est bien celle du destinataire concerné et non une contrefaçon. Il contient des informations associées à la clé publique d'une personne, aidant d'autres personnes à vérifier qu'une clé est authentique ou valide.

Un certificat numérique X.509 [23] se compose de trois éléments :

- une clé publique,
- des informations sur le certificat. (Informations sur l'identité de l'utilisateur,

telles que son nom, son ID utilisateur, etc.),

- une ou plusieurs signatures numériques d'une autorité de certification.

La signature numérique du certificat permet de déclarer que les informations qu'il contient ont été attestées par une autre personne ou entité. La signature numérique ne garantit pas totalement l'authenticité du certificat. Elle confirme uniquement que les informations d'identification signées correspondent ou sont liées à la clé publique. Ainsi, un certificat équivaut en réalité à une clé publique comportant un ou deux types d'ID joints ainsi qu'une estampille agréée par d'autres personnes fiables.

2.8 Conclusion

En se dématérialisant, l'argent devient information, donc mémorisable et contrôlable. L'exploitation des informations disponibles sur l'Internet met à la portée des entreprises, des individus, des services secrets et des malfaiteurs des données personnelles. Le développement du commerce électronique conduit de la sorte à d'avantage de contrôle et d'espionnage.

Toute transaction commerciale repose sur la confiance mutuelle des intervenants en leur probité, dans la qualité des biens échangés et dans la fiabilité des systèmes de transfert de paiement ou de livraison des achats. Puisque les échanges associés au commerce électronique se déroulent la plupart du temps à distance, un climat de confiance est nécessaire à la conduite des affaires.

Les solutions de sécurité réalisant les services de sécurité, l'authentification, l'identification, la confidentialité et l'intégrité, se basent en grande partie sur la cryptographie en utilisant des algorithmes d'encodage spécifiques. Pour la réalisation de l'authentification et de l'identification, on propose des mécanismes mettant en oeuvre une signature numérique avec un algorithme à clé publique et des certificats numériques délivrés par des autorités de certification assermentées. Pour couvrir les risques liés à la confidentialité, on propose des mécanismes chiffrant les messages avec des algorithmes à clé publique, s'il s'agit de messages courts, parce qu'ils présentent l'avantage d'une sûreté élevée ou des algorithmes à clé secrète, s'il s'agit de messages longs, parce qu'ils sont plus rapide que les algorithmes à clé publique, ils présentent cependant l'inconvénient de l'échange de clé secrète qu'on peut résoudre par l'utilisation d'un

Chapitre 3

SecAdvise

SecAdvise [22, 21] est un aviseur de sécurité conçu pour être interopérable dans tout système d'échanges électroniques. Il a l'habileté de sélectionner, et ce d'une manière dynamique et automatique, les mécanismes ou les systèmes de sécurité nécessaires pour exécuter une transaction de commerce électronique entre un certain nombre de participants souhaitant communiquer d'une manière sûre.

Dans ce chapitre, nous commençons par rappeler le modèle de confiance [20] sur lequel SecAdvise est basé et la formule de sélection préliminaire de cet aviseur en se basant sur [22, 21]. Nous proposerons ensuite une extension du modèle en introduisant deux nouvelles notions qui sont la réduction du risque et le coût. Enfin nous introduisons une éventuelle base de données pour SecAdvise.

3.1 Un modèle de confiance pour les applications e-commerce

Le modèle de confiance sur lequel se base SecAdvise [22, 21] décrit une méthodologie pour dériver une solution qui s'applique aux différents problèmes de confiance. Ce modèle, décrit dans l'article [20], définit un espace de problèmes de confiance qu'il divise en des sous-espaces de problèmes indépendants.

3.1.1 Définitions [20]

L'espace de problèmes de confiance (TPS : *Trust problem space*) est défini comme étant l'ensemble de toutes les situations dans le système dans lesquelles les agents

e-commerce peuvent avoir des problèmes de confiance entre eux ou avec l'environnement, par exemple : attaques, vulnérabilités, etc.

Les sous-espaces de problèmes de confiance (TPSS : *Trust problem sub-space*) sont le résultat de la division du TPS en des sous-espaces indépendants. Un TPSS est décrit formellement comme suit :

TPSS ***(Id, situation, conséquence, niveau de l'impact, probabilité, intervalle de temps, localisation)***

Id l'identificateur du TPSS ;

situation la situation qui cause le problème (une seule situation par TPSS) ;

conséquence l'impact du problème s'il survient, peut être une liste de conséquences ;

niveau d'impact catastrophique, critique, marginal ou négligeable ;

probabilité avec laquelle le problème peut survenir avec un niveau d'impact catastrophique, critique, marginal ou négligeable ;

intervalle de temps durant laquelle la situation peut durer ;

localisation du problème dans le système, chaque TPSS doit être bien localisé géographiquement dans le système. On peut avoir plusieurs endroits.

Une unité de confiance (*TU : trust unit*) est une unité logique qui représente une solution ou une contre-mesure partielle ou complète d'un TPSS. Elle peut être un protocole cryptographique, un mécanisme de contrôle ou une infrastructure comme celle des clés publiques.

Les composantes d'une TU sont :

TU ***(Id, description,*** **TPSS**, ***type, mécanisme, dépendance, durée)***

Id un identificateur unique pour chaque TU ;

description fait référence aux concepts de confiance couverts par ce TU ;

TPSS le ou les TPSS(s) concerné(s) par ce TU ;

type la nature de ce TU par rapport à chaque TPSS concerné par ce TU est soit :

dissuasif réduit le risque ou l'impact du problème,

préventif protège et bloque l'impact d'une éventuelle attaque ou

correctif réduit les problèmes causés par le TPSS.

mécanisme le mécanisme utilisé par le TU qui est l'implémentation de l'aspect de confiance proposé par ce TU pour couvrir partiellement ou en entier un TPSS. Ce mécanisme peut être un protocole ou une infrastructure ;

dépendance le TU peut faire partie d'une structure plus grande et donc ce champs sert à donner la liste des autres TUs avec lesquels il est lié ou qui utilisent ce TU ;

durée à quel moment du cycle de vie de l'application, le TU est appliqué.

Une unité de confiance doit être la plus simple possible, sa taille dépend du niveau de granularité du modèle de confiance. Une solution de confiance (*TS : Trust Solution*) est un ensemble d'unités de confiance qui couvre totalement l'espace de problèmes de confiance. Il peut y avoir plusieurs solutions.

L'objectif de* SecAdvise *est de soumettre le choix d'un ensemble de*TU *à un ensemble de critères qui vont déterminer lequel des ensembles est la solution la mieux adaptée aux attentes des participants.

3.1.2 La méthodologie du modèle de confiance

L'idée sur laquelle se base la méthodologie est de définir un ensemble de contraintes pour améliorer la confiance. Cela comprend les services de sécurité classiques : confidentialité, authentification, non-répudiation des actions et des informations, et les mécanismes de contrôle pour améliorer la performance du système. La méthodologie comprend trois étapes :

1. définir le TPS en définissant les besoins de confiance et les vulnérabilités ;
2. définir les unités de confiance représentant les solutions partielles pour les TPSSs ;
3. choisir et combiner une partie de ces TUs pour composer un ensemble de solutions qui couvre tout le TPS ; une seule solution doit être sélectionnée. Cette sélection peut prendre en compte le nombre d'agent concernés, les parties externes ou les infrastructures disponibles.

Pour la réalisation de cette méthode, on doit créer une librairie dynamique générale pour l'ensemble des TUs, TPSSs et TSs. Cette librairie peut être utilisée pour définir

des modèles de confiance pour d'autres systèmes. Aussi pourra-t-on définir quelques recommandations pour concevoir des solutions de confiance. On doit aussi créer un formalisme logique pour rendre automatique le processus de sélection des unités de confiance d'une manière optimale afin de produire la meilleure solution de confiance possible.

3.2 Un aviseur de mécanismes de sécurité : SecAdvise

À la base, l'idée derrière la conception de SecAdvise [22, 21] était de fournir un environnement qui intégrera les mécanismes de sécurité et qui fournira dynamiquement une solution de confiance satisfaisant les contraintes de sécurité exigées par les parties souhaitant exécuter une transaction sans risque. Un tel aviseur projette de surmonter les problèmes de compatibilité et d'interopérabilité, de réduire les risques technologiques de sécurité et d'augmenter la confiance des utilisateurs dans les systèmes de commerce électronique.

Nous reprenons les notations de SecAdvise définies dans [22, 21] et qui sont :

c la transaction à sécuriser. C'est le contexte/transaction à exécuter et qui a besoin d'être sécurisé ;

$\mathbf{U}$ l'ensemble de toutes les Unités de confiance TU (Trust Unit). Une TU peut être un mécanisme de sécurité, un protocole de sécurité, ou une infrastructure de sécurité. Un élément pourraît être décomposé en sous éléments, (par exemple SSL) ;

u une unité de confiance ($u \in \mathbf{U}$) ;

$\mathbf{R}$ l'ensemble de tous les risques de sécurité non décomposables, tel que $\forall\, r \in \mathbf{R}$, $\forall\, u \in \mathbf{U}$ u couvre entièrement r ou bien u ne couvre pas r ;

r un risque de sécurité non décomposable ($r \in \mathbf{R}$) ;

$\mathbf{P}$ l'ensemble de tous les participants potentiels dans une transaction sécurisée ;

p un participant ($p \in \mathbf{P}$) ;

R_u l'ensemble des risques de sécurité couverts par l'unité de confiance u ($R_u \in \mathcal{P}(\mathbf{R})$) ;

R_c l'ensemble des risques de sécurité à couvrir dans le contexte/transaction c ($R_c \in \mathcal{P}(\mathbf{R})$). Ce sont les TPSSs définis au-dessus ;

P_c l'ensemble de tous les participants qui sont impliqués directement dans le contexte/transaction c. ($P_c \in \mathcal{P}(\mathbf{P})$), par exemple, un client veut payer à un marchand ;

$A_{u,p}$ l'ensemble des participants auquels le participant p fait confiance, et qui peuvent jouer le rôle d'une autorité certifiante dans un mécanisme de sécurité ou une TU u. ($u \in \mathbf{U}$, $p \in \mathbf{P}$, $A_{u,p} \in \mathcal{P}(\mathbf{P})$). Si le TU n'a pas besoin d'une troisième partie de confiance, $A_{u,p} = \mathbf{P}$, pour simplifier le processus d'association entre les unités de confiance ;

U_p l'ensemble des unités de confiance disponibles à un participant $p \in \mathbf{P}$ ($U_p \in \mathcal{P}(\mathbf{U})$) ;

$\bar{U}_P$ l'ensemble des unités de confiance disponibles à tous les participants $\forall p \in \mathbf{P}$ ($\bar{U}_P \in \mathcal{P}(\mathbf{U})$)

$$\bar{U}_P = \{u \in \bigcap_{p \in P} U_p \quad | \quad \bigcap_{p \in P} A_{u,p} \neq \emptyset\} ;$$

$\tilde{U}_c$ l'ensemble minimal d'unités de confiance couvrant les risques de sécurité de la transaction/contexte c ($\tilde{U}_c \in \mathcal{P}(\mathbf{U})$)

$$\tilde{U}_c = U \in \mathcal{P}(\bar{U}_{P_c}) \quad | \quad R_c \subseteq \bigcup_{u \in U} R_u \wedge ||U|| = \min_{U' \in \mathcal{P}(\bar{U}_{P_c})} ||U'||.$$

L'ensemble $\mathbf{R}$ [22, 21] a été défini dans la version préliminaire de SecAdvise comme étant l'ensemble des risques non décomposables tel que pour n'importe quel $u \in \mathbf{U}$, u couvre entièrement r ou u ne couvre nullement r.

Cette hypothèse de conception est utopique et prendrait tout son sens si pour chaque risque de sécurité, il existait une solution qui le réduit à 100%, alors que la théorie de la cryptanalyse nous a démontré que tout algorithme de cryptographie

peut être cassé même si cela prendra plusieurs années de calcul avec des machines très puissantes.

Vu que la réalité ne coïncide pas avec cette hypothèse, nous introduirons dans la section suivante de nouvelles notions à SecAdvise :

1. le pourcentage de couverture d'un risque r par une unité de confiance u,
2. les contraintes régissant l'exécution d'une transaction/contexte c et
3. le facteur coût de cette couverture.

3.3 Sélection de mécanismes

Selon Robles [20], tout problème de sécurité (TPS) peut être décrit ou défini par un ensemble de sous-espaces de problèmes TPSSs mutuellement exclusifs. Aussi selon Robles, pour chaque sous-espace de problème TPSS, il existe un TU ou un ensemble de TU permettant de couvrir le TPSS. Selon cette logique, la solution à tout espace de problème TPS est un ensemble de TUs.

Dans cette section, nous définissons une méthode de sélection de protocoles de sécurité ou d'un ensemble de mécanismes de sécurité qu'on va appeler *patron* qui offre une solution *optimale* pour un problème de sécurité particulier. Le patron utilisera une base de données pour déterminer un ensemble d'unités de confiance pouvant représenter une solution globale.

Partant du principe qu'un problème peut avoir plusieurs solutions, le patron doit choisir une seule solution qui sera jugée optimale en tenant compte de contraintes additionnelles qui ne sont pas forcément liées aux risques propres à la sécurité. Les participants cherchant à effectuer une transaction peuvent être confrontés à des problèmes de nature économique ou politique : puissance de machines et serveurs, limitation de budget, contraintes politiques, interdiction sur certaines formes de cryptographie, contraintes de temps, etc. Nous proposons donc que le patron ne prenne pas juste en compte l'ensemble des TPSSs représentant le problème à résoudre mais aussi une série de paramètres représentant les contraintes qui limitent le choix des solutions possibles.

Ainsi, l'ajout d'une table de contraintes à la base de données de SecAdvise que nous définirons dans la section 3.5, s'impose. Du fait qu'il peut y avoir plusieurs solutions

à un TPSS, le nombre de solutions pour un TPS peut être extrêmement grand. Pour trier entre ses solutions et trouver une solution optimale, nous proposons quelques principes de base :

- une solution unique est toujours préférable à une solution composite.
  Par exemple : une solution à un seul TU pour couvrir un TPSS est préférable à une union de TU ;
- une solution avec le moindre coût pour l'utilisateur ;
- on donne priorité aux solutions validées par une méthode de vérification des protocoles cryptographiques (Espaces de *Strand* [7], PROMELA [11], etc.) ;
- on prend une solution qui réduit au maximum l'ensemble des risques couverts par l'ensemble des TUs.

3.4 Couverture de risque, contraintes et coût

Soient les notations suivantes pour les nouveaux éléments à ajouter à SecAdvise :

C l'ensemble des contextes/transactions traitées par SecAdvise jusqu'à présent,

$$(c \in \mathbf{C})$$

$\$_c$ le coût au delà duquel la transaction c ne peut être exécutée. C'est une somme multifactorielle du coût en termes économiques, temporel, politique que peut supporter chacun des participants dans la transaction c, SecAdvise doit communiquer avec les participants de la transaction afin de calculer ce coût.

CON l'ensemble de toutes les contraintes qu'elles soient économiques, temporelles, politiques, etc.

con une contrainte économique, temporelle, politique, etc.

$$(con \in \mathbf{CON})$$

CON_c l'ensemble des contraintes qui régissent la transaction c.

$$CON_c \in \mathcal{P}(\mathbf{CON})$$

% une fonction qui calcule le pourcentage de réduction d'un risque r par une unité de confiance u sous une contrainte $con \in \mathbf{CON}$.

$$\% : \mathbf{U} \times \mathbf{R} \times \mathbf{CON} \longrightarrow [0,1]$$
$$(u, r, con) \longmapsto \%(u, r, con)$$

$ une fonction qui calcule le coût de la couverture d'un risque r par une unité de confiance u sous un sous-ensemble de contraintes $CON \in \mathcal{P}(\mathbf{CON})$.

$$\$: \mathbf{U} \times \mathbf{R} \times \mathcal{P}(\mathbf{CON}) \longrightarrow R$$
$$(u, r, CON) \longmapsto \$(u, r, CON)$$

$\%(r, u, CON)$ la réduction d'un risque r par une unité de confiance u contrainte à un sous-ensemble de contraintes CON ($CON \in \mathcal{P}(\mathbf{CON})$)

$$\%(r, u, CON) = \min_{con \in CON} \%(r, u, con)$$

$\%_{r,c}$ l'ensemble des réductions du risque r dont le coût ne dépasse pas celui de c ($\$_c$)

$$\%_{r,c} = \{ \%(r, u, CON_c) \mid u \in U \wedge \$(r, u, CON_c) \leq \$_c \}$$

$U_{r,c}$ l'ensemble des unités de confiance qui maximisent l'ensemble $\%_{r,c}$

$$U_{r,c} = \{u \in U \mid \%(r, u, CON_c) = max(\%_{r,c}) \neq 0\}$$

$\bar{U}_c$ l'ensemble des solutions composites maximisant la réduction du coût sous les contraintes CON_c tout en minimisant le coût de la transaction

$$\bar{U}_c = \{ U \in \mathcal{P}(\bar{U}_{P_c}) \mid \forall r \in R_c \, \exists u \in U$$
$$(u \in U_{r,c}) \wedge (\$(r, u, CON_c) = \min_{u' \in U_{r,c}} \$(r, u', CON_c)) \}$$

$\tilde{U}_c$ l'ensemble d'unités de confiance optimal qui couvre au maximum R_c avec le moindre coût. R_c est l'esemble des risques de sécurité du contexte/transaction c qui est soumis à un ensemble de contraintes ($\tilde{U}_c \in \mathcal{P}(\mathbf{U})$)

$$\tilde{U}_c = U \in \bar{U}_c \mid \|U\| = \min_{U' \in \bar{U}_c} \|U'\|$$

3.5 Une base de données pour SecAdvise

Afin de permettre à SecAdvise de sélectionner une solution optimale suivant les critères établis dans ce chapitre, nous avons défini une base de données, qui d'une part

servira de structure de sauvegarde des données et paramètres de tout contexte de transaction, et d'autre part gardera la trace de déroulement de l'exécution de SecAdvise à chaque fois que ce dernier est sollicité. Cela permettra à SecAdvise d'améliorer ses sélections ultérieures en se servant des calculs déjà entrepris.

En se basant sur les définitions du TPS, du TPSS et du TU du modèle de *Robles* [20] et des notions nouvellement introduites, nous avons défini les tables composant la base de données. Leur contenu est listé dans le tableau 3.1. Pour garder le lien avec le modèle théorique de SecAdvise décrit ci-haut, nous avons gardé les mêmes notations pour la définition des tables de la base de données.

3.6 Conclusion

Dans ce chapitre, nous avons amélioré la définition de SecAdvise, un outil de sélection de mécanismes de sécurisation des transactions électroniques, par l'introduction de la notion du coût, des contraintes et de validation. Il s'agit du coût du déroulement de la transaction sous des contraintes économiques, politiques et temporelles. À partir de ces notions, nous avons calculé un ensemble de jointures d'unités de confiance optimales en termes de réduction des risques et des coûts, puisque tous les éléments de cet ensemble réduisent au même degré les risques et les coûts alors SecAdvise peut choisir aléatoirement une solution de cet ensemble. Quant à la validation, nous lui consacrons les chapitres qui suivent. Il s'agit de la dernière étape que SecAdvise doit entreprendre pour valider, de préférence automatiquement, la solution avant de la mettre en exécution. Comme nous allons le montrer dans les chapitres suivants, la validation consiste en une preuve que l'union des unités de confiance couvre réellement l'ensemble des risques menaçant le contexte/transaction en question.

Nom de la table	*Attributs*	*Type de donnée*
R	Identificateur_r	DECIMAL
	Situation	TXT
	Niveau_Impact	SMALLINT
	Probabilité	DECIMAL
	Intervalle_Temps	TIME
Conséquences	ID_conséquence	DECIMAL
	Conséquence	TXT
r_Conséquence	ID_conséquence	DECIMAL
	Identificateur_r	DECIMAL
Localisation	ID_Localisation	DECIMAL
	Localisation	TXT
r_Localisation	ID_Localisation	DECIMAL
	Identificateur_r	DECIMAL
U	Identificateur_u	DECIMAL
	Description	TXT
	ID_Mécanisme	TXT
C	Identificateur_c	DECIMAL
	Description	TXT
	$\$c$	FLOAT
CON	Identificateur_con	DECIMAL
	Type	VARCHAR
$\tilde{U}_c$	ID_Solution	DECIMAL
	description	TXT
	Valide	BOOLEAN
	Méthode_validation	TXT
Units_solution	ID_Solution	DECIMAL
	Identificateur_u	DECIMAL
Domaine_Application	ID_Domaine	DECIMAL
	Description	TXT
Units_Domaine_Application	ID_Domaine	DECIMAL
	Identificateur_u	DECIMAL
C_R	Identificateur_r	DECIMAL
	Identificateur_c	DECIMAL
C_CON	Identificateur_con	DECIMAL
	Identificateur_c	DECIMAL
CON_U	Identificateur_u	DECIMAL
	Identificateur_con	DECIMAL
R_U	Identificateur_u	DECIMAL
	Identificateur_r	DECIMAL
	Nature	VARCHAR
	$\%(r,u)$	DECIMAL
	$\$(r,u)$	VARCHAR

TAB. 3.1 – Contenu des tables de la base de données de SecAdvise

Chapitre 4

Modèle de validation

Dans le chapitre précédent, nous avons ajouté la contrainte de validation des solutions à l'aviseur SecAdvise. Ainsi, avant de proposer une combinaison d'unités comme solution pour couvrir un ensemble de risques, SecAdvise doit effectuer trois opérations de vérification :

1. la vérification de chaque unité faisant partie de la solution en validant sa spécification d'une manière formelle ce qui aidera à l'automatisation de la validation globale de la solution ;
2. la validation de l'union des unités, prétendant couvrir un ensemble de risques dans un environnement donné. Ceci revient à prouver que la jointure des mécanismes de sécurité (les unités de confiance) est exempte de toute faille intrinsèque et que les propriétés ou services de sécurité de la solution sont vérifiés ;
3. la détection d'intrusion en simulant un modèle de validation de la solution avec un modèle d'intrus.

Ceci nous pousse à nous poser les questions suivantes :

- *Comment valider ou prouver qu'une unité de confiance prétendant couvrir un risque répond effectivement à son engagement ?*

  Cette question est légitime étant donné que même en transformant l'hypothèse qu'avec une très haute probabilité, seul le détenteur de la clé peut obtenir en clair le texte chiffré, en une certitude (l'hypothèse du chiffrement parfait), les attaques par usurpation d'identité abondent [3]. Les aspects à vérifier sont de type logique. C'est pour cela qu'on les appelle ***aspects logiques des protocoles cryptographiques***. Le fameux exemple du protocole *Needham-Schroeder* [14]

à clé publique, employé pendant plus de quinze ans (et soi-disant prouvé) avant qu'on en trouve une attaque, en témoigne.

- *Étant donné deux unités de confiance u_1 et u_2 couvrants respectivement deux risques disjoints r_1 et r_2, est-ce que l'union $u_1 \bigcup u_2$ couvre effectivement $r_1 \bigcup r_2$?*

  Ceci revient à se demander si l'introduction d'un autre mécanisme ne met pas en danger l'efficacité du premier.

  Le problème se pose parce que l'union $u_1 \bigcup u_2$ est en fait un nouveau protocole et l'élaboration d'un nouveau protocole passe toujours par l'étape de la validation pour corriger toute faille dans les règles de procédures (*procedures rules*).

- *Si on arrive à trouver d'une façon ou d'une autre des failles de type « aspects logiques », est-ce possible de les corriger ?*

  La correction automatique de telles erreurs n'est pas toujours possible. En effet, il existe des failles qui mettent en cause toute la conception du protocole de sécurité et donc leur correction revient à revoir toute la conception des régles de procédures. Par contre, il existe des erreurs qui peuvent être corrigées automatiquement ou manuellement sans toucher à la structure générale du protocole, par exemple une erreur touchant une partie du déroulement et qui est indépendante des autres étapes.

Dans ce chapitre, nous définissons ce qu'est un modèle de validation de manière générale, nous abordons ensuite les machines à états finis étendues avec lesquelles on peut présenter le comportement d'un protocole de sécurité, nous expliquons la logique temporelle linéaire (LTL) qui sert à exprimer les critères de corrections et finalement nous présentons le langage PROMELA/SPIN que nous avons choisi pour valider notre cas d'étude.

4.1 Structure d'un protocole de sécurité

La spécification d'un protocole de sécurité consiste à spécifier explicitement les cinq parties distinctes qui le définissent et qui sont :

1. le service à être fourni par le protocole,
2. les hypothèses sur l'environnement dans lequel le protocole est exécuté,
3. le vocabulaire utilisé pour implémenter le format des messages du protocole,
4. l'encodage de chaque message échangé dans ce vocabulaire et
5. les règles qui régissent l'échange des données et qui assurent la cohérence du protocole, celle-ci étant la partie la plus difficile à spécifier.

La spécification d'un protocole, en général, et plus spécifiquement d'un protocole de sécurité peut être comparée à la définition d'un langage. En effet, le format du protocole équivaut à la syntaxe du langage, les règles du protocole jouent le rôle de la grammaire du langage à une différence prêt, c'est que la grammaire d'un langage peut être ambiguë alors que celle d'un protocole ne doit pas l'être. C'est même une condition pour juger de la cohérence d'un protocole. Enfin, la spécification du service fourni par le protocole peut être comparée à la sémantique du langage.

4.2 Modèle de validation des protocoles de sécurité

Nous avons présenté ci-dessus les cinq éléments qui entrent dans la définition d'un protocole de sécurité. Pour valider un protocole, il suffit donc de trouver un moyen de spécifier les cinq éléments le définissant. Les quatre premiers éléments sont faciles à spécifier du fait qu'ils ne sont pas régis par des changements d'état ; leur spécification est statique et on peut la faire par n'importe quel langage de programmation. Ce n'est pas le cas des règles qui régissent l'échange des données au sein du protocole, faisant basculer l'état du système de part et d'autre vers un autre état d'une manière déterministe mais dynamique.

Automatiser la validation des protocoles de sécurité nécessite la définition d'un langage formel dans lequel on peut spécifier le comportement de tels protocoles. La spécification formelle d'un protocole sert comme entrée pour un outil de vérification automatique. Il existe plusieurs langages de spécification pouvant servir à cette fin. On notera entre autres SDL, ESTELLE, LOTOS et PROMELA.

Pour notre étude de cas, nous avons choisi le langage PROMELA (*Protocol Meta Language*) pour les raisons suivantes :

- la simplicité de sa syntaxe qui est très proche du langage C,
- l'existence d'un paquetage déjà disponible SPIN (*Simple Promela Interpreter*) qui permet la simulation d'un protocole de communication concurrent à partir d'un code PROMELA,
- les détails de l'implémentation n'ont pas à être tous spécifiés parce que c'est la cohérence des règles du protocole qui est visée dans la validation et non son implémentation qui inclut les algorithmes de cryptographie.

Avec l'outil SPIN qui interprète le code PROMELA, on peut simuler le comportement d'un protocole distribué ainsi que celui d'un intrus à l'environnement d'exécution du protocole à valider en lui procurant tout les pouvoirs dont use un intrus réel dans un système ouvert.

Dans la section qui suit, nous introduisons le langage PROMELA que nous avons utilisé dans notre étude de cas pour la spécification du comportement d'un protocole de sécurité dans un modèle de validation formel.

4.3 PROMELA

PROMELA (*Protocol Meta Language*) [1] est un langage de spécification formelle basé sur les machines à états communicantes.

C'est un langage non déterministe basé sur la notation du langage de Dijkstra [6], le *Guarded Command Language*. Pour les opérations d'entrée/sortie, il emprunte la notation du langage CSP (*Communicating Sequential Processes*) de *Hoares* qui est une notation informelle des opérateurs de base de la concurrence distribuée.

PROMELA a la capacité de créer dynamiquement des processus concurrents. Il permet aussi la communication entre les processus via des canaux de messages de manière synchrone (rendez-vous entre processus) ou asynchrone (tampons).

Un programme PROMELA est essentiellement composé de processus, de canaux de messages et de variables. Les processus spécifient le comportement des parties communicantes alors que les variables et les canaux définissent l'environnement dans

[1] http ://spinroot.com/spin/man/quick.html

lequel les processus s'exécutent. PROMELA possède cinq classes d'unités lexicales, qui peuvent être séparés par des blancs, tabulations, retours à la ligne et commentaires. Ces cinq classes sont :

1. les identificateurs,
2. les mots-clés,
3. les constantes,
4. les opérateurs et
5. les séparateurs d'instructions.

On trouve les types d'instruction suivants :

- assignation et condition,
- sélection et répétition,
- envoie et réception de message,
- goto et break,
- timeout (permet à un processus d'abandonner l'attente d'une condition qui ne peut jamais devenir vraie).

PROMELA possède trois types d'objet :

- processus,
- canaux,
- variables.

Par définition, tous les processus sont des objets globaux. Les variables et les canaux représentent des données qui peuvent être globales ou locales à un processus. Dans PROMELA, il n'y a pas de différence entre conditions et instructions. En effet, l'exécution d'une instruction est conditionnelle à son *exécutabilité*. Toutes les instructions de PROMELA sont soit exécutées, soit bloquées, dépendamment des valeurs courantes des variables et du contenu des canaux de messages. L'exécutabilité est la notion de base de la synchronisation dans PROMELA. Un processus peut donc attendre qu'un événement se produise pour qu'une instruction devienne exécutable. Comme on va le voir plus bas, l'assignation des variables est toujours exécutable. Considérons par exemple l'instruction en C suivante :

```
/*attendre pour a == b
*/
```

```
while (a != b) skip;
```

en PROMELA l'instruction serait :

```
/*l'exécution est bloquée jusqu'à ce que a == b
*/
(a == b)
```

Les opérateurs arithmétiques et booléens ont la même syntaxe que celle du langage C.

4.3.1 Les variables et les types de données

Dans PROMELA, les variables peuvent être globales, et concernent donc le protocole de sécurité à modéliser en entier, ou locales à un processus, et donc seul ce processus, a accès à cette valeur (par exemple une clé privée). On peut utiliser six types de données prédéfinis : bit, bool, byte, short, int, chan.

chan spécifie les canaux où sont déposés les messages servant à la communication de données entre les différents processus. Une variable de type chan est un objet qui peut contenir plusieurs valeurs groupées dans une structure définie par l'utilisateur.

4.3.2 Le type **proctype**

En PROMELA, le processus modélise des machines à états finis. Il est défini par une partie en-tête (nom, paramètres et déclaration) et un bloc d'instructions. Pour déclarer un processus, on utilise le mot clé proctype comme dans l'exemple suivant :

```
proctype Identifiant_proc (liste_paramètres) {
    Séquences d'instructions
}
proctype A() {
    byte state;
    state = 3
}
```

On a donné le nom A au processus qu'on vient de créer. Le corps de la déclaration se trouve entre les deux accolades { } et contient la déclaration des variables locales au processus A et les instructions.

Le « ; » est un séparateur d'instructions. Il ne joue pas le rôle d'un marqueur de fin

d'instructions, ceci explique l'absence du « ; » après la deuxième instruction.
En fait, PROMELA possède deux séparateurs d'instructions : le point virgule « ; » et la flèche « → », les deux sont équivalents. On utilise la flèche pour indiquer informellement qu'il existe une relation causale entre deux instructions. La partie se trouvant à gauche de la flèche est appelée prédicat.
Dans l'exemple suivant :

(state == 1) → state = 3

l'exécution est bloquée sur le test de la condition booléenne state == 1 jusqu'à ce qu'elle soit vraie, ensuite l'assignement de la variable state par la valeur 3 est exécuté.

Une définition proctype déclare le comportement d'un processus mais ne l'exécute pas. C'est le processus de type init qui est exécuté au départ et qui peut lancer l'exécution d'un ou de plusieurs processus à l'aide de l'instruction run.

Programme 1 *init et run*

```
byte state = 2;
proctype A() {
    (state == 1) → state = 3
}
proctype B() {
    state = state − 1
}
init{
    run A();
    run B()
}
```

Dans cet exemple, le processus init lance l'exécution du processus A et puis celle de B et se termine.
L'opérateur unaire run instancie une copie d'un type de processus donné.
run est toujours exécutable à moins que le nombre de processus instanciés n'ai atteint la limite maximale que peut supporter le système.

L'opérateur run peut passer des paramètres de type : bit, bool, byte, short, int aux processus, comme dans le programme qui suit :

Programme 2 *passage de paramètres à run*

```
proctype A(byte state; short set) {
    (state == 1) → state = set
}
init{
    run A(1,3)
}
```

Les variables globales posent le problème de l'exclusion mutuelle que PROMELA peut résoudre par la déclaration de séquences atomiques. L'exemple qui suit illustre cette capacité.

Programme 3 *séquence atomic*

```
proctype A() {
    atomic{
        (state == 1) → state = state + 1
    }
}
proctype B()  {
    atomic{
        (state == 1 ) → state = state − 1
    }
}
init{
    run A();
    run B()
}
```

La séquence des instructions préfixée par atomic ne peut être entrelacée avec d'autres instructions appartenant à d'autres processus, mais est plutôt exécutée en une seule unité indivisible.

4.3.3 Les canaux de messages

Les canaux de messages sont utilisés pour modéliser le transfert de données d'un processus à un autre. Ils peuvent être déclarés globaux ou locaux. Par exemple :

```
chan qname = [16] of {
    byte,
    int,
    chan,
    byte
}
```

déclare un canal qname qui peut stocker 16 messages, chaque message possédant quatre champs avec les types byte, int, chan et byte, respectivement. On ne peut stocker qu'une seule structure de données par canal.
La lecture et l'écriture se font en FIFO (*First In First Out*). L'instruction qname ! exp écrit la valeur de l'expression expr à la fin du canal qname tandis que l'instruction qname ? expr lit la valeur se trouvant à la tête du canal et la stocke dans la variable expr.

En plus de la capacité de spécifier formellement le comportement d'un certain nombre de participants dans une solution de sécurité avec le langage PROMELA, on peut aussi spécifier explicitement les critères qui permettent de juger qu'une telle solution est correcte. La section 4.4 aborde ces critères qui sont appelés *critères de corrections* et qui diffèrent d'une solution à l'autre.

4.3.4 Les flux de commandes

En plus de la concaténation des instructions dans un processus, l'exécution parallèle des processus et les séquences atomiques, PROMELA permet trois autres flux de commandes :

La sélection elle se fait à l'aide des mots clés if et fi comme dans l'exemple suivant :

```
if
:: (a != b) → option1
:: (a ==b) → option2
fi
```

Les valeurs des variables a et b servent à choisir entre deux options. La structure de sélection contient deux séquences d'exécution, chacune précédée par un double deux-points. Une seule séquence de la liste est exécutée. La séquence peut être sélectionnée seulement si sa première instruction est exécutable. La première instruction de chaque séquence est appelé *guard*.

Si plus d'un *guard* est exécutable alors une seule séquence est choisie aléatoirement pour être exécutée. Si tous les *guards* sont non-exécutables alors le processus bloque jusqu'à ce qu'au moins un *guard* devienne exécutable. Il n'y a aucune restriction quant au type d'instruction pouvant être utilisée comme *guard*. Dans l'exemple qui suit on utilise une lecture et une écriture comme *guard* :

```
if
:: can?a
:: can!b
fi
```

On peut même utiliser des assignations comme *guard* :

```
if
:: compt = compt + 1
:: compt = compt - 1
fi
```

La répétition Une extension de la sélection est la répétition qui est définie à l'aide des mots clés do et od comme dans le programme suivant où la variable compt est incrémentée et décrémentée d'une manière aléatoire :

```
byte compt;
proctype compteur()
{
    do
    :: compt = compt + 1
    :: compt = compt - 1
    :: (compt == 0) → break
    od
}
```

À chaque répétition, une seule séquence peut être sélectionnée. Après que l'exécution de la séquence choisie ait terminée, l'exécution de la boucle est répétée. Pour sortir de la boucle on utilise un break. Dans l'exemple, on sort de la boucle lorsque le compteur est nul (compt == 0).

Le saut inconditionnel L'instruction goto est une autre façon de sortir de la boucle. L'implémentation suivante de l'algorithme d'Euclid pour trouver le diviseur commun de deux nombres positifs illustre l'utilisation de do/od et goto :

```
proctype Euclid(int x, y) {
```

```
    do
    :: (x > y) → x = x - y
    :: (x < y) → y = y - x
    :: (x == y) → goto done
    od;
    done :
          skip
}
```

skip est toujours exécutable et n'a aucun effet. goto est une instruction toujours exécutable dans PROMELA.

4.4 Les critères de correction

Les programmes qui peuvent être écrits en langage comme PROMELA sont appelés ***modèles de validation***. Pour valider une conception (union d'unités de confiance dans notre cas), nous devons être capable de spécifier précisément ce que signifie une conception correcte. On doit donc définir des critères de correction spécifiques. Il existe trois critères standards [11] :

- l'absence d'inter-blocages (*deadlocks*). Un inter-blocage est une situation qui provoque un blocage complet du système et qui se produit lorsque deux processus tentent d'avoir accès à une même ressource simultanément, ou lorsque l'un d'entre eux attend une information que seul l'autre peut lui fournir, mais est temporairement dans l'impossibilité de le faire ;
- l'absence de *livelocks*. Un *livelock* est l'exécution d'une séquence qui se répète indéfiniment sans aucun progrès (ou évolution) ;
- l'absence de terminaisons impropres. Une terminaison impropre survient à la fin de l'exécution d'un protocole sans satisfaire les conditions de terminaisons propres à chaque conception.

4.4.1 Formalisation des critères de correction dans PROMELA

Invariants du système

Les critères de correction peuvent dans la majorité des cas être exprimés comme des expressions booléennes qui doivent être satisfaites à chaque fois que le processus

atteint un état donné. Dans PROMELA, on peut exprimer ces critères à l'aide d'un processus du type prédéfini monitor et de l'instruction assert.

```
proctype monitor () {
    assert (condition)
}
```

Une fois une instance du processus monitor lancée, le validateur SPIN peut décider d'évaluer l'assertion à n'importe quel moment. En effet, l'instruction assert est exécutable exactement une fois pour chaque état du système. Dans une vérification exhaustive, SPIN, l'outil de vérification utilisé avec PROMELA, doit parcourir tous les états du système et vérifier si la condition est vraie. S'il existe au moins une seule séquence d'exécution dans laquelle la condition est fausse, alors le critère pour lequel est assigné cette assertion est violé.

États finaux propres

En PROMELA, on peut identifier les états d'un processus donné comme étant des états finaux propres en les étiquetant par end comme suit :

```
end :
do
:: canal1 ! p → canal2 ? v
od
```

Un état final dans une séquence d'exécution terminée est un état final propre si :

- chaque processus, qui a été instancié a terminé son exécution ou bien a atteint un état marqué par l'étiquette end et
- tous les canaux de communications sont vides.

Livelocks

Deux propriétés de séquences cycliques peuvent être exprimées en PROMELA. Les deux propriétés sont basées sur le marquage explicite des états dans un modèle de validation :

1. l'étiquette progress marque un état qui doit être exécuté pour marquer un progrès pour le protocole et

2. l'étiquette accept marque un état qui ne peut pas faire partie d'une séquence d'états pouvant se répéter infiniment.

Critères temporels

Les critères temporels définissent les exigences dans l'ordre des états d'un système. Par exemple, pour exprimer le critère suivant :
« *Chaque état dans lequel la propriété P est vraie est suivi obligatoirement par un état dans lequel la propriété Q est vraie* »
dans le langage PROMELA nous pouvons écrire :
$P \rightarrow Q$

Alors que pour exprimer un ordre temporel qui ne doit pas avoir lieu nous pouvons écrire :

```
never {
    ...
}
```

où les points contiennent les détails du critère.

4.5 Machines à états finis étendues

Un protocole de sécurité peut être représenté comme une *machine à états finis* [2]. La conception des critères de correction devient plus facile vu qu'ils peuvent être exprimés comme des états désirables ou pas. L'état du protocole symbolise les hypothèses que chaque processus dans le système fait à propos des autres processus. Il définit quelles actions le processus va entamer, quels événements sont attendus et comment réagir à ces événements. Les règles qui définissent une solution de sécurité sont elles aussi dans la majorité des cas modélisables par une machine à états finis.

La variante des *machines à états finis étendues* [11] utilisée pour modéliser un modèle de validation PROMELA est définie formellement comme suit :
Un tuple $(Q,\ q_0,\ M,\ A,\ T)$, où

Q est un ensemble d'états non vide,

[2] Structure dans laquelle le calcul est modélisé sous la forme d'une transition d'un état vers un autre parmis un nombre fini d'états, aussi appelée automate d'états finis.

q_0 est un élément de Q, il représente l'état initial de la machine,

M est l'ensemble des canaux de messages par lesquels les machines communiquent,

A est l'ensemble des noms des variables,

T est la relation de transition d'états.

La relation T prend deux arguments, $T(q, a)$, où q est l'état courant et a est une action. $T(q, a)$ est l'état vers lequel la machine transite quand elle se trouve à l'état q et entreprend l'action a. Les actions permises dans les machines à états finis étendues sont :

- les *inputs* et les *outputs*. Les actions d'*Input/Output* sont définies comme étant des ensembles de valeurs finis et ordonnés. Ces valeurs peuvent être des variables de l'ensemble A, ou simplement des constantes. Par définition, la première valeur de cet ensemble ordonné définit le canal de destination de l'I/O, ce canal appartient à l'ensemble M. Le reste des valeurs définit une structure qui sera ajoutée ou enlevée du canal quand l'action I/O sera exécutée ;
- l'action nulle ϵ (son équivalent en langage PROMELA est l'instruction skip) ;
- les conditions booléennes. PROMELA définit une condition exécutable seulement si elle est vraie ;
- les assignations des éléments de l'ensemble A. Une assignation peut changer la valeur d'une seule variable. Les expressions sont composées de variables de l'ensemble A, des valeurs constantes et des opérateurs arithmétiques et relationnels. Rappelons que dans PROMELA une assignation est toujours exécutable.

Il y a une équivalence entre un programme PROMELA et une machine à états finis étendue. L'exemple qui suit montre cette équivalence.
Soit le programme PROMELA :

Programme 4 *Euclid*

```
chan In, Out;
proctype Euclide {
    int x, y;
    In ? x,y;
```

```
    do
    :: (x > y) → x = x − y
    :: (x < y) → y = y − x
    :: (x == y) break
    od;
    Out ! x
}
```

Le processus de type Euclide commence par recevoir deux valeurs x et y et il termine son exécution en écrivant le plus grand diviseur commun de ces deux valeurs dans le canal de sortie Out. La machine à états finis étendue qui est équivalente à ce code est définie dans le tableau 4.1, où la colonne *Action* regroupe les conditions, les assignations et les opérations I/Os.

Les composantes de l'automate sont :

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, où

- q_0 est l'état initial de l'automate qui correspond à l'étape avant l'éxecution de la première instruction du code
- q_1 est l'état vers lequel transite le système après la lecture par le processus Euclide du canal In
- q_2 est l'état du système lorsque les deux valeurs se trouvant à la tête du canal In vérifie l'inégalité x > y,
- q_3 est l'état du système lorsque les deux valeurs se trouvant à la tête du canal In vérifie l'inégalité x < y,
- q_4 est l'état du système lorsque les deux valeurs se trouvant à la tête du canal In vérifie l'égalité x == y,
- q_5 est l'état du système lorsque le processus Euclide écrit dans le canal Out.

$M = \{In, Out\}$,

$A = \{x, y\}$,

Actions l'ensemble des actions utilisées dans ce modèle sont :

- les actions de type I/O, In ? x,y et Out ! x,
- les actions de type condition booléenne : x == y, x > y et x < y,
- les actions de type assignement de variables : x = x − y et x = y − x.

T À partir de la définition des états q_0, q_1, q_2, q_3, q_4 *et* q_5 présentée ci-dessus, on définit la relation de transition T telle qu'elle est présentée dans le tableau 4.1 :

- $T(q_0, In\ ?\ x, y) = q_1$,
- $T(q_1, x > y) = q_2$,
- $T(q_1, x < y) = q_3$,
- $T(q_1, x == y) = q_4$,
- $T(q_2, x = x - y) = q_1$,
- $T(q_3, y = y - x) = q_1$,
- $T(q_4, Out\ !\ x) = q_5$,
- $T(q_5, -) = -$.

État courant	*Action*	État suivant
q0	In ? x,y	q1
q1	x > y	q2
q1	x < y	q3
q1	x == y	q4
q2	x = x-y	q1
q3	y = y-x	q1
q4	Out ! x	q5
q5	-	-

TAB. 4.1 – Une machine à états finis étendue

4.6 Une logique temporelle linéaire (LTL)

Pour prouver qu'un programme satisfait une propriété, SPIN utilise le modèle de vérification LTL (*Linear Temporal Logic*) [2]. Quand une propriété est exprimée à l'aide d'une formule LTL, SPIN transforme la négation de cette formule en un automate de *Buchi*, construit le produit (l'intersection) de cet automate avec le programme, et vérifie que l'intersection est vide. Si ce n'est pas le cas, alors la propriété n'est pas respectée par le programme.

LTL [2] a été introduit pour spécifier les propriétés de l'exécution d'un système. Un ensemble fini *Prop* contient toutes les propriétés atomiques des états du système. Avec les opérateurs booléens ($\neg$, $\wedge$, $\vee$), nous pouvons exprimer seulement des propriétés

statiques. Pour les propriétés dynamiques, LTL utilise des opérateurs temporels tels que X (*next*), U (*until*), R (*release*), F (*eventually*) et G (*always*).

Syntaxe :

L'ensemble des formules LTL bâties sur l'ensemble *Prop* est défini par la grammaire $\varphi ::= p \mid \neg \varphi \mid \varphi \vee \varphi \mid X \varphi \mid \varphi U \varphi$, où p parcours l'ensemble *Prop*.

Sémantique :

La sémantique de LTL définit si une exécution σ d'un système donné satisfait une formule. La sémantique dépend des propositions atomiques qui appartiennent à chaque état de σ. Soit $\Sigma = 2^{Prop}$ l'ensemble des parties de *Prop* et soit $\sigma = \sigma_0\sigma_1 \ldots$ un mot de Σ^ω, ç'est à dire, une séquence infinie d'éléments de Σ.

Soit φ une formule LTL. La relation $\sigma \models \varphi$ (σ modélise φ) est définit comme suit :

- $\sigma \models p$ *si* $p \in \sigma_0$,
- $\sigma \models \neg\varphi_1$ *si* $\sigma \not\models \varphi_1$,
- $\sigma \models \varphi_1 \vee \varphi_2$ *si* $\sigma \models \varphi_1$ *ou* $\sigma \models \varphi_2$,
- $\sigma \models X \varphi_1$ *si* $\sigma_1\sigma_2 \ldots \models \varphi_1$,
- $\sigma \models \varphi_1 U \varphi_2$ *si* $\exists k \geq 0$, $\sigma_k\sigma_{k+1} \ldots \models \varphi_2$ *et* $\forall 0 \leq i < k$, $\sigma_i\sigma_{i+1} \ldots \models \varphi_1$.

LTL met aussi à notre disposition des opérateurs dérivés des opérateurs de base, ils sont définis par :

- $tt \equiv p \vee \neg p$,
- $ff \equiv \neg tt$
- $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$,
- $\varphi_1 R \varphi_2 \equiv \neg(\neg\varphi_1 U \varphi_2)$,
- $F \varphi \equiv tt U \varphi$ et
- $G \varphi \equiv ff R \varphi = \neg F \neg\varphi$.

Toute formule LTL qui n'est ni disjonctive ($\vee$) ni conjonctive ($\wedge$) est appelée une *formule temporelle.*

Une formule LTL peut être écrite en *forme normale négative*, en utilisant seulement les prédicats de *Prop*, leur négation, et les opérateurs $\vee$, $\wedge$, X, U, et R.

4.7 SPIN (Simple Promela INterpreter)

SPIN [11] est un outil pour l'analyse des systèmes distribués et plus spécifiquement des protocoles communicant des données. Le système à évaluer par SPIN doit être modélisé dans le langage de spécification formelle PROMELA.

SPIN peut jouer le rôle d'un simulateur et d'un évaluateur.

Un simulateur en simulant le comportement d'un système distribué.

Un évaluateur en analysant d'une manière exhaustive, aléatoire ou guidée des violations comme :

- l'existence d'une séquence qui échoue,
- l'existence d'états finaux invalides,
- l'existence de boucles infinies sur des états désirables,
- la satisfaction de propriétés exprimées en LTL.

XSPIN est une interface graphique de SPIN (écrite en tcl/tk) qui permet la visualisation de machines à états, de MSCs *(Message Sequence Charts)*[3], etc.

SPIN peut être utilisé en trois modes de base :

1. en mode simulation, SPIN peut simuler l'exécution d'un modèle de validation en interprètant *à la volée*[4] les instructions de PROMELA. Il peut être utilisé pour une conception partielle ou complète d'un protocole, à n'importe quel niveau d'abstraction. Utiliser SPIN en mode simulation permet un prototypage rapide avec trois options possibles : une simulation aléatoire, guidée ou interactive.

   La figure 4.1 montre la structure générale du simulateur ; il est composé des

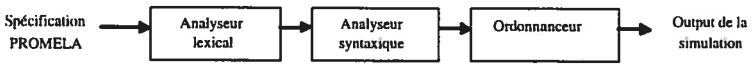

FIG. 4.1 – Structure générale du simulateur SPIN

éléments suivants :

[3]Diagramme de séquences représentant l'exécution d'un système composé de plusieurs processus qui communiquent par des canaux fiables (http ://www.granddictionnaire.com).

[4](*On-the-fly*) Se dit de ce qui peut effectuer une tâche, ou être effectué, sur-le-champ et rapidement, de façon automatique, sans interruption (http ://www.granddictionnaire.com).

(a) un analyseur lexical qui est utilisé pour reconnaître les terminaux et les littéraux,

(b) un analyseur syntaxique qui vérifie la grammaire et construit un arbre syntaxique, et

(c) un ordonnanceur qui évalue le programme en parcourant l'arbre et en évaluant ses nœuds en accordance avec la sémantique du langage. L'arbre syntaxique est connecté de manière à ce que l'ordonnanceur puisse l'évaluer à la volée ;

2. en mode vérificateur (ou validateur) exhaustif, SPIN peut évaluer un espace d'états de manière exhaustive pour prouver la validité des critères de correction imposés par le concepteur d'un protocole (par défaut les *deadlocks*). Il utilise à cette fin la théorie de réduction à ordre partiel pour optimiser sa recherche. Cette validation ne peut être utilisée que si les ressources matérielles le permettent (puissance de CPU et mémoire disponible) ;

3. en mode *supertrace*, SPIN évalue les espaces d'états de systèmes qui exigent une taille plus grande de mémoire pour leur évaluation. Il utilise une technique d'optimisation du parcours de l'espace d'états qui donne d'excellentes *couvertures d'erreurs*[5] avec une mémoire limitée. Plus précisément, ce sont des techniques de réduction par ordre partiel qui consistent à identifier les chemins qu'il doit explorer pour garantir la vérification la plus complète possible de chaque propriété de correction.

La figure 4.2 illustre la structure générale du validateur SPIN. Une procédure appelée run() alloue la mémoire et prépare toutes les structures de données que le validateur utilisera durant sa recherche. Elle appelle la procédure new_state() pour effectuer la recherche en cours. Les deux structures de données utilisées par cette procédure sont l'espace d'état et une matrice de transition qui encode en entier le modèle de validation écrit en PROMELA. Chaque instruction du modèle possède une entrée dans cette matrice ; elle définit le prédicat de l'exécutabilité de l'instruction ainsi que l'effet de son exécution.

[5] couverture d'erreurs = le nombre d'erreurs distinctes trouvées divisé par le nombre total d'erreurs présentes dans le système

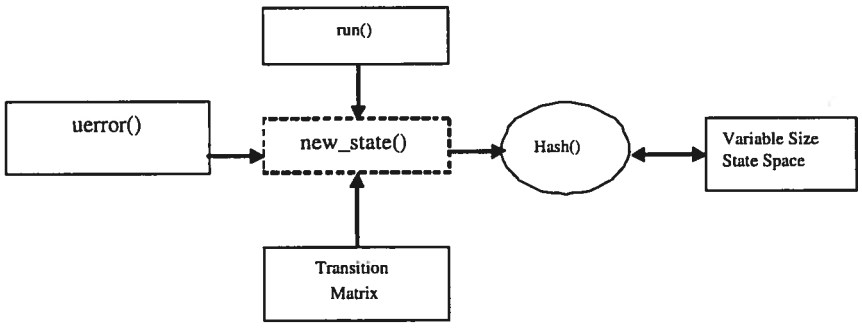

FIG. 4.2 – Structure générale du validateur SPIN

L'état actuel du système est maintenu dans un vecteur de valeurs qui peut augmenter et diminuer de taille de manière dynamique. La procédure new_state() effectue une recherche en profondeur de toutes les instructions exécutables dans le modèle. Au lieu de choisir une seule instruction exécutable de la liste des processus en cours, comme le fait le simulateur, le validateur a pour tâche de tester les effets de toutes les instructions exécutables à un état donné du système avec tous les entrelacements possibles.

Avant de passer à l'analyse d'un nouvel état, la fonction new_state() décide quel état a déjà été analysé et pourra donc être sauté. S'il trouve une erreur, la procédure uerror() est appelée pour produire une trace du chemin menant à l'erreur trouvée et par défaut arrête l'analyse.

Une erreur peut être une violation des critères de correction, par exemple, une assertion violée ou un état du système où tous les processus sont bloqués.

Parmi les caractéristiques de SPIN, on peut citer :

- SPIN est utilisé pour tracer les erreurs de type logique dans la conception de systèmes distribués comme les systèmes d'exploitation, les protocoles de communication de données, les systèmes de commutation dans les télécommunications, les algorithmes concurrents, etc. L'outil vérifie la cohérence logique d'une spécification. Il reporte les critères de correction imposés par le concepteur ;
- SPIN fonctionne à la volée, ce qui veut dire qu'il permet la construction d'un graphe d'état global, ou d'une structure de *Kripke*[6], comme entrée à une vérifica-

[6]Un automate étiqueté par des symboles qui représentent les propriétés élémentaires vérifiées par chaque état

tion de n'importe quel ensemble de propriétés d'un système donné ;

- SPIN peut être utilisé comme un vérificateur complet de système modélisé en LTL (*linear temporal logic*), supportant toute propriété de correction exprimée en LTL ;
- SPIN supporte dynamiquement un nombre croissant ou décroissant de processus ;
- SPIN supporte trois techniques de communication entre processus : par rendez-vous (communication synchrone), par passage de messages via des tampons (communication asynchrone) et la communication via mémoire partagée ;
- SPIN exploite la technique de réduction à ordre partiel pour optimiser l'exécution de la vérification.

4.8 Conclusion

Dans ce chapitre, nous avons introduit les définitions des notions mises en jeu par les modèles de validation. Nous avons aussi présenté le langage de vérification formelle PROMELA et l'outil SPIN servant à la simulation et à la vérification automatique des modèles de validation écrits en PROMELA.

Dans le chapitre qui suit, nous réalisons un cas d'étude basé sur la validation du protocole de sécurité SSL avec l'outil PROMELA/SPIN.

Chapitre 5

Étude de cas : validation d'une solution

Nous avons abordé dans les chapitres précédents l'importance du critère de validation dans le choix d'une solution par SecAdvise, comme nous avons proposé l'utilisation de l'outil PROMELA/SPIN pour la validation de notre étude de cas. Dans ce chapitre, nous allons démontrer l'importance de cette validation par un exemple, en expliquant chaque étape dans ce processus.

5.1 Mise en situation

Notre étude de cas consiste en la validation de l'union de quatre unités de confiance choisies par SecAdvise pour couvrir un ensemble de risques de sécurité. Nous nous mettons dans la situation où deux participants, un serveur et un client, veulent assurer un échange de données de manière à ce qu'ils s'authentifient l'un auprès de l'autre, que le contenu des données échangées reste confidentiel, que les données échangées arrivent intègres à leur destination et qu'il y ait un moyen pour prouver la non-répudiation de part et d'autre.

On peut relever les quatre risques de sécurité suivants à couvrir dans ce contexte/transaction :

r_1 risque d'interception de messages secrets,

r_2 risque d'usurpation de l'identité d'un participant,

r_3 risque d'altération du contenu du message transmis,

r_4 risque de dénis de transaction.

On suppose que SecAdvise, après avoir effectuer tous les calculs nécessaires (coût de la transaction, pourcentage de réduction maximale des risques du contexte/transaction) a sélectionné les unités de confiance suivantes :

u_1 le protocole SSL pour couvrir $r_1 \cup r_2 \cup r_3$,

u_2 une signature numérique pour couvrir r_4.

En fait, u_1(SSL) est une union de trois mécanismes de sécurité atomiques, soit la cryptographie pour couvrir r_1, un certificat numérique pour couvrir r_2 et une fonction de hachage pour couvrir r_3. On a $u_1 = \{u'_1, u'_2, u'_3\}$.
On note les deux participants de cette transaction client et serveur. On a client, serveur $\in P$.
Notons le contexte/transaction à sécuriser c. On a :

- $r_1, r_2, r_3, r_4 \in R$,
- $u_1 \in \mathcal{P}(\mathbf{U})$, $u_2 \in \mathbf{U}$,
- $r_1, r_2, r_3 \in R_{u_1}$ et $r_4 \in R_{u_2}$,
- $R_c = \{ r_1, r_2, r_3, r_4 \}$,
- $P_c = \{ client, serveur \}$,
- Pour simplifier le modèle de validation, on suppose que le client et le serveur ont le même ensemble $A_{u,p}$, l'ensemble des autorités de certification auquelles ils font confiance. On suppose donc que $A_{u,serveur} = A_{u,client}$,
- Comme notre objectif n'est pas de démontrer pourquoi SecAdvise aurait choisi u_1 et u_2 comme solution à ce contexte/transaction, nous ne nous arrêterons pas sur les calculs qu'il aurait effectués. On suppose donc sans le prouver que $\tilde{U}_c = u_1 \cup u_2$.

Nous nous concentrons sur les étapes par lesquels nous sommes passés afin de spécifier formellement la solution et les critères de correction afin de les valider avec l'outil SPIN. Dans le chapitre 6 nous nous attarderons sur le processus de l'intrus qui constitue un élément clé dans la validation de toute solution de sécurité.

5.2 Modélisation de SSL

5.2.1 Couverture des risques par SSL

Comme il a été exposé dans la section 2.6.1 du chapitre 2, SSL couvre les trois risques r_1, r_2 et r_3 en fournissant les trois services de sécurité suivants :

la confidentialité de la connexion en utilisant le chiffrement symétrique. Parmis les algorithmes que SSL peut négocier, on trouve DES, RC4, etc. ;

l'identification et l'authentification du client et du serveur. SSL utilise à cette fin la cryptographie à clé publique. Les algorithmes pouvant être utilisés sont : RSA, DSS, etc ;

l'intégrité des données. SSL ajoute aux messages échangés un condensât pour la vérification de l'intégrité (MAC). Les fonctions de hachage, SHA ou MD5 sont utilisées pour le calcul de ce MAC.

SSL prend les messages à transmettre, les fragmente en des bloques de données, les compresse, leur applique un MAC, les chiffre, et transmets les résultats à l'autre point de la communication. Les messages reçus par l'autre paire sont alors décryptés, vérifiés, décompressés et rassemblés.

5.2.2 Le format des messages échangés dans SSL

La figure 5.1 illustre les messages échangés entre un client et un serveur pendant l'établissement d'une nouvelle session. Les messages échangés, suivant leur ordre chronologique, sont :

HelloRequest ce message demande au client d'entamer le *Handshake* ;

ClientHello ce message contient

- le numéro de version du protocole SSL,
- le nombre aléatoire *client_random*,
- l'identificateur de session *session_ID*,
- la liste des suites de chiffrement choisies par le client et
- la liste des méthodes de compression choisies par le client.

ServerHello ce message contient :

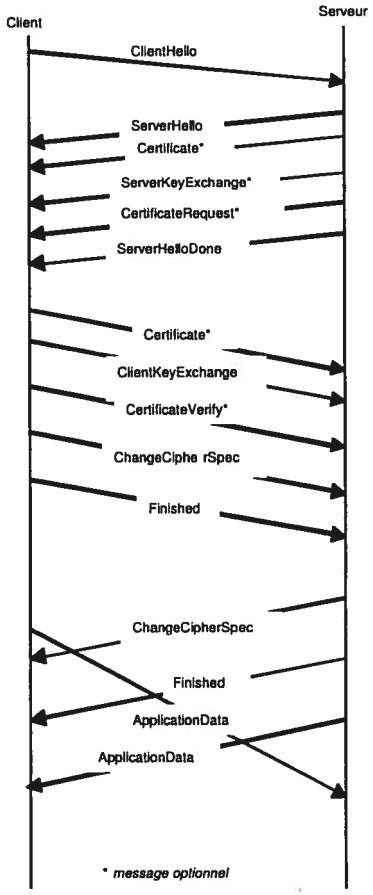

FIG. 5.1 – Messages échangés pendant l'etablissement d'une nouvelle session SSL

- le numéro de la version du protocole SSL,
- un nombre aléatoire *serveur_random*,
- l'identificateur de session *session_ID*,
- une suite de chiffrement et
- une méthode de compression.

Certificate ce message contient le certificat du serveur ou celui du client si le serveur le lui réclame et que le client en possède un. Le certificat doit être un certificat X.509 version 3.0 qui renferme la clé publique reliée à la suite de chiffrement choisie par le message *ServerHello*. Rappelons qu'un certificat X.509 est sous la forme suivante :

$Certificat =$

$$Nom\_Entite + Nom\_AC + KP\_Entite +$$
$$KS\_AC\{H[Nom\_Entite + Nom\_AC + Kp\_Entite]\}$$

Où :

Nom_Entite est le nom du serveur ou du client dépendamment qu'il s'agisse du certificat du serveur ou du client,

NOM_AC est le nom de l'autorité de certification,

KP_Entite est la clé publique de l'entité à certifier,

$KS\_AC\{x\}$ est le sceau de x calculé avec la clé privée de l'autorité de certification,

$H[y]$ est le condensât de y ;

ServerKeyExchange ce message n'est envoyé par le serveur que s'il ne possède aucun certificat, ou seulement un certificat de signature ;

CertificateRequest par ce message, le serveur réclame un certificat au client ;

ServerHelloDone ce message signale la fin de l'envoi des messages *ServerHello* et subséquents ;

ClientKeyExchange ce message contient le *PreMasterSecret* crypté à l'aide de la clé publique du serveur ;

CertificateVerify ce message permet une vérification explicite du certificat du client. En effet, si le client est certifié et que le certificat lui confère la faculté

d'apposer une signature numérique, il envoie également le message de confirmation explicite *CertificateVerify*. Ce message dont le but est de permettre au serveur de vérifier le sceau du client, contient le condensât de tous les messages depuis *ClientHello*, à l'exception du message conteneur (*CertificateVerify*) ; ce condensât est chiffré avec la clé privée du client.

Ainsi le contenu du message *CertificateVerify* sera :

$Hash\{MasterSecret + hash(messages\_deja\_envoyes + MasterSecret)\}$

où *hash* est l'algorithme MD5 ou SHA selon le cas ;

Finished ce message signale la fin du protocole *Handshake* et le début de l'émission des données protégées avec les nouveaux paramètres négociés. C'est un condensé de tous les messages de *Handshake* que le client a déjà envoyés au serveur depuis *ClientHello* en employant les attributs cryptographiques qui viennent d'être négociés. On déjoue ainsi les attaques de subtilisation en vérifiant l'intégrité de l'ensemble des échanges. Le condensât est calculé à l'aide de la formule suivante :

$$Hash\{MasterSecret + hash(handshake\_messages + Sender + MasterSecret)\}$$

Où *hash* est MD5 ou SHA selon le cas. Le champ *handshake_messages* contient les messages échangés sauf le message *ChangeCipherSpec* et le message courant. Notons la présence du champ *Sender* qui contient l'identité de l'expéditeur.

À la réception du message *Finished*, le serveur tente de reproduire le même condensât à partir des messages précédemment reçus. Il compare le résultat au contenu du message *Finished* qu'il vient de recevoir du client. Cette étape lui permet de déceler si un intrus a intercepté et modifié les messages. Le serveur envoie à son tour les messages *ChangeCipherSpec* et *Finished*. Là encore, le message *Finished* est généré à partir de tous les messages que le serveur a précédemment envoyés et il est transmis chiffré. Le serveur commence aussitôt à envoyer ses données d'application.

5.2.3 Opérations cryptographiques

L'échange de clés, l'authentification, l'encodage, et les algorithmes pour le calcul du *MAC* sont déterminés par la suite de chiffrement (*cipher_suite*) sélectionnée par le serveur et révélée au client dans le message *ServerHello*.

Chiffrement asymétrique

Des algorithmes asymétriques sont utilisés dans le protocole *Handshake* pour authentifier les parties et pour générer les clés secrètes. Des fonctions à sens unique sont utilisées pour chiffrer les données sortantes. Les données chiffrées avec la clé publique d'une paire de clés donnée ne peuvent être déchiffrées que par la clé privée.

Pour Diffie-Hellman, RSA, et Fortezza, le même algorithme est utilisé pour convertir le *pre_master_secret* en *master_secret*. Il est calculé comme suit :

$Master\_secret$ =

$$MD5(pre\_master\_secret + SHA(pre\_Master\_secret + ClientHello.random + ServerHello.random).$$

Chiffrement symétrique

SSL peut négocier deux types de chiffrement symétrique :

enfilé (appelé aussi en continu ou à la volée) qui consiste à convertir le texte en clair bit par bit en combinant la suite de bits du texte en clair et la série de bits de la clé de chiffrement à l'aide d'un *ou exclusif* ;

en bloc qui agit par transformation de blocs de données de taille fixe en blocs chiffrés de même taille.

Signature digitale

Des fonctions de hachage à sens unique servent comme entrée pour les algorithmes servant à la signature. Dans la signature avec l'algorithme RSA par exemple, une structure de deux *hashs* (une calculée à l'aide de la fonction de hachage SHA et l'autre à l'aide de MD5) est signée (chiffrée avec une clé privée). Alors que dans la signature avec l'algorithme DSS, la structure du *hash* est calculée à l'aide de SHA seulement.

Par exemple, dans le message suivant :

$Stream\_ciphered(field1, field2, digitally\_signed(hash))$

le contenu de *hash* sert comme entrée pour l'algorithme de signature, ensuite toute la structure est cryptée par un algorithme de chiffrement symétrique de type enfilé.

Définition des canaux

Les canaux sont le moyen de communication des processus entre eux. Nous avons défini un type de canal pour chaque format de message utilisé par SSL. Puisque les messages de SSL sont sous une des formes suivantes :

- *HelloRequest*{*serveur*},
- *ClientHello*{*client, client_hello*},
- *ServerHello*{*serveur, server_hello*},
- *Certificate*{*client (ou serveur), certificat_client ou (certificat_serveur),* {*certificat_client*}*KS*{*client*} *(ou* {*certificat_serveur*}*KS*{*serveur*}*)* },
- *CertificateRequest*{*serveur, certificate_request*},
- *ServerHelloDone*{*serveur, server_hello_done*},
- *ClientKeyExchange*{{*pre_master_secret*}*KP*{*serveur*}},
- *CertificateVerify*{*client,* {{*handshake_messages*}*HASH*}*PS*{*client*} } ou
- *Finished* {*client (ou serveur),* {*handshake_messages, client (ou serveur), masterSecret*}*HASH*}.

et comme nous voulons exiger le fait qu'un message est toujours échangé entre un participant légitime (le client ou le serveur) et l'intrus, ce dernier ayant la possibilité d'acheminer le message à la bonne destination, alors nous spécifierons toujours au début de chaque message l'identité du participant concerné dans la transmission. L'autre bout n'a pas besoin d'être spécifié puisque c'est forcément l'intrus. On aura donc besoin des canaux suivants :

```
chan HelloRequest = [0] of {mtype};
chan ClientHello = [0] of {mtype, mtype};
chan ServerHello = [0] of {mtype, mtype};
chan Certificate = [0] of {mtype, mtype, mtype, mtype};
chan CertificateRequest = [0] of {mtype, mtype};
chan ServerHelloDone = [0] of {mtype, mtype};
chan ClientKeyExchange = [0] of {mtype, mtype};
chan CertificateVerify = [0] of {mtype, mtype};
chan Finished = [0] of {mtype, mtype};
chan NoCertificate = [0] of {mtype};
chan ChangeCipherSpec = [0] of {mtype}.
```

ils doivent être déclarés globaux pour que tous les processus y accèdent.

Par exemple, si le processus du *serveur* veut envoyer le message *PK(client){message}*, l'instruction suivante doit être utilisée :

```
c2pk ! serveur, message, client
```

où *serveur* représente celui qui envoie le message, *message* et *client* représentent le contenu du message envoyé (client définit uniquement *PK(client)*). La réception d'un message par un participant est exprimée similairement par :

```
c2pk ? eval(serveur), x1, eval(client).
```

Cette instruction n'acceptera comme valide que les messages avec comme receveur *serveur* (ce test sert à s'assurer que le message est bel et bien adressé à *serveur*) et avec la valeur du troisième champ *client*. On notera $x1$ est une variable locale où sera stocké le deuxième paramètre.

Définition du processus serveur et client

Les processus représentant les différents participants dans SSL doivent être paramétrables avec les données qui changent d'une session à l'autre ou d'une instance à l'autre. Leur définition doit aussi obligatoirement inclure les conditions qui représentent les règles qui régissent le protocole et qui doivent être vérifiée ou validées. Comme nous l'avons démontré dans le chapitre précédent, l'implémentation d'un EFSM dans le langage PROMELA peut se faire d'une manière automatique. À partir des EFSMs décrivant le comportement d'un serveur de SSL, représenté sur la figure 5.2, et celui d'un client représenté sur la figure 5.3, nous avons généré le code PROMELA des processus *serveur* et *client*. Ce code est disponible à l'annexe A. Le paramètre *self* représente l'identité du client, alors que *party* représente celle du serveur. Les séquences précédées par *atomic* sont utilisées pour réduire le taux d'entrelacements ce qui réduit la complexité du modèle du point de vue de la vérification.

5.2.5 Les critères de correction

Validation de l'authentification

Nous avons défini des macros pour mettre à jour les prédicats qui expriment les propriétés d'authentification du protocole SSL. Cette technique présentée dans [15]

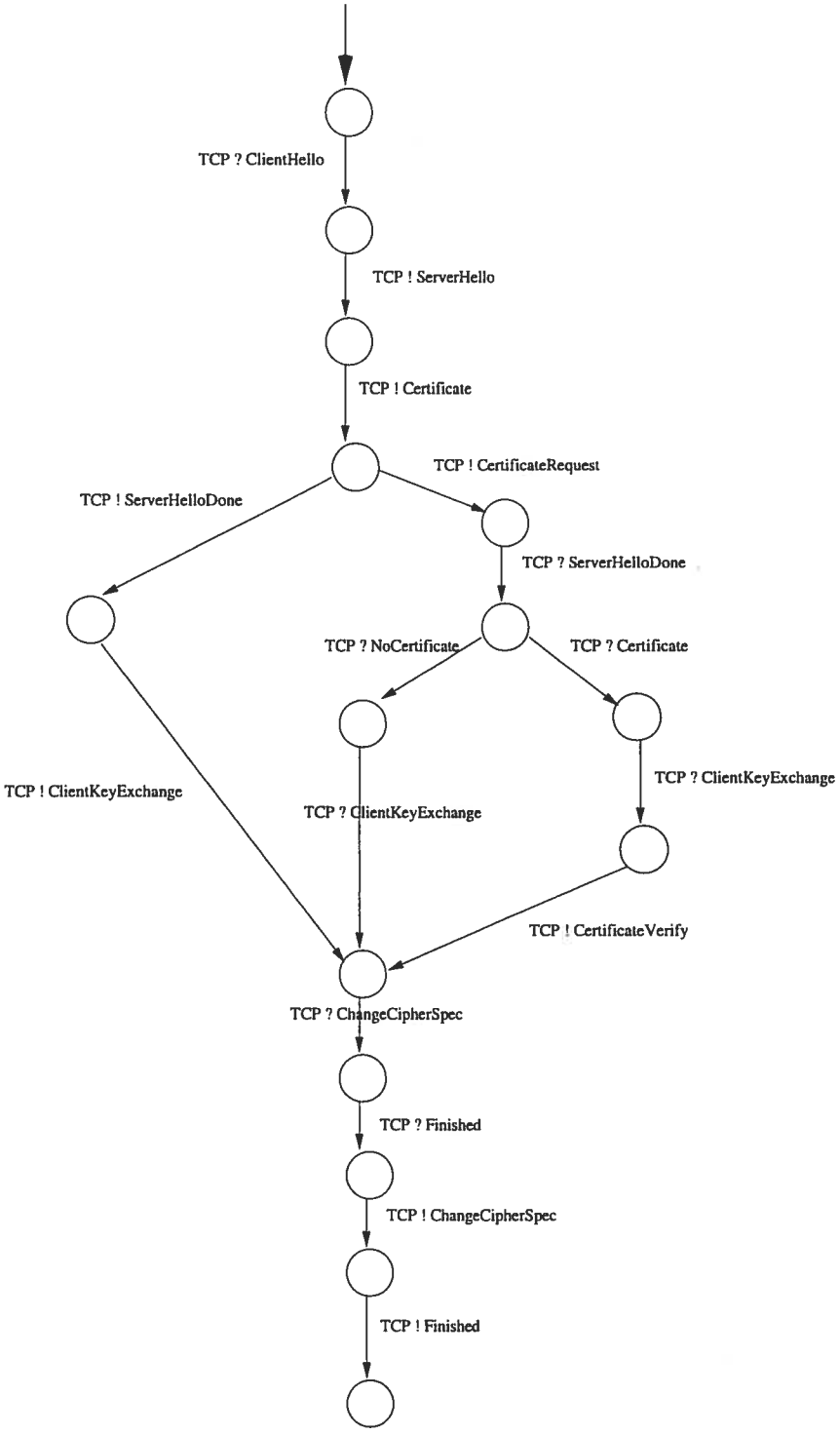

FIG. 5.2 – EFSM représentant le comportement d'un serveur SSL/signature numérique

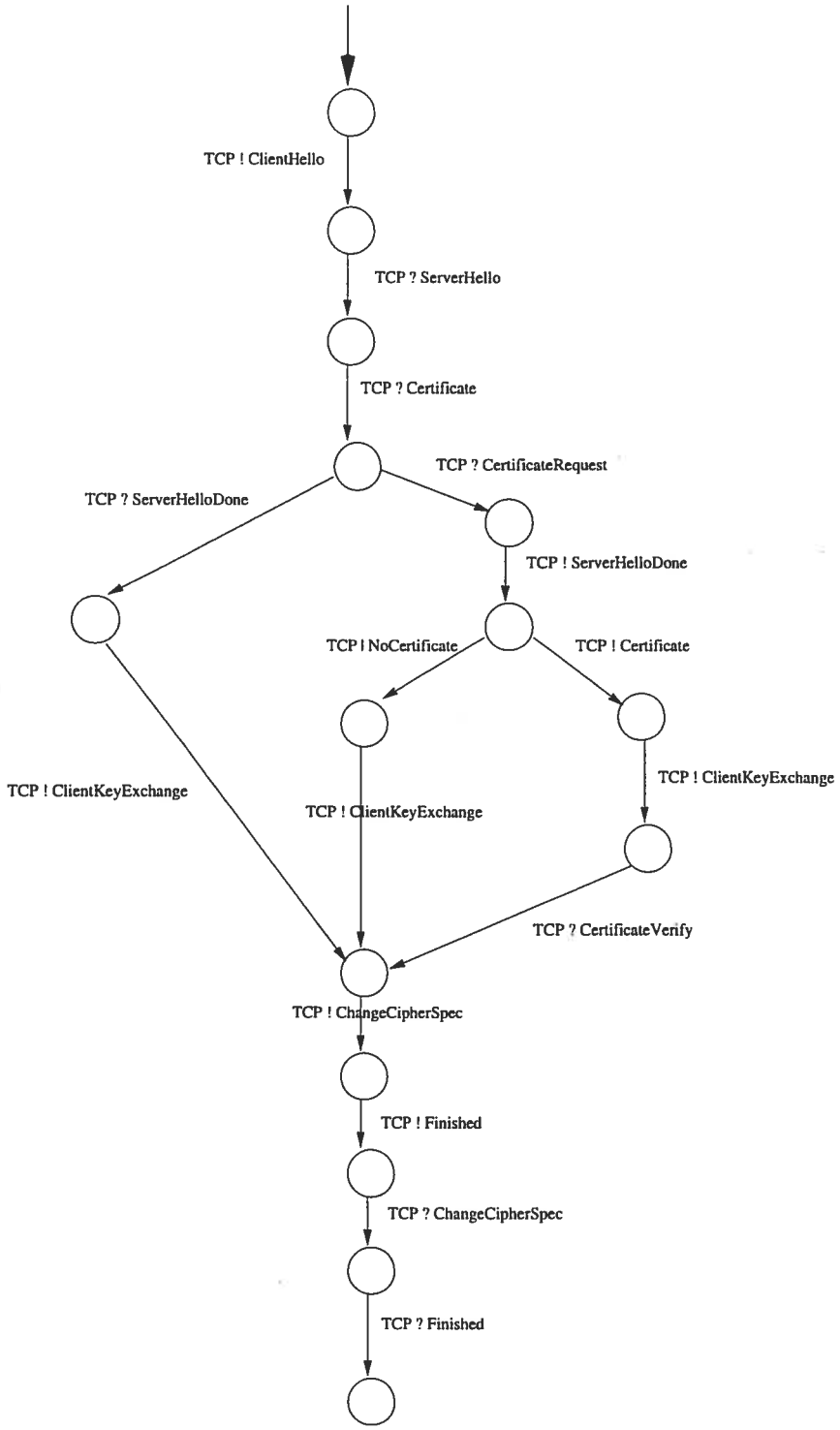

FIG. 5.3 – EFSM représentant le comportement d'un client SSL/signature numérique

se base sur deux définitions. La première définit un agent X comme une entité prenant part dans une instance de protocole avec l'agent Y s'il initie une session de ce protocole avec Y. La deuxième définit un agent X comme une entité commettant une session avec Y dans une instance d'un protocole si X a correctement terminé une session de ce protocole avec Y.

Le fait qu'un serveur avec l'identité *serveur* s'authentifie correctement auprès d'un client avec l'identité *client* peut être exprimé par la proposition suivante :
client commet une session avec *serveur* seulement si *serveur* a en effet pris part dans une instance du protocole avec *client*.
Une proposition similaire exprime la propriété réciproque c'est-à-dire le fait qu'un client avec l'identité *client* s'authentifie auprès d'un serveur avec l'identité *serveur*.

Chacune de ces propositions peut être représentée dans PROMELA par une variable globale qui devient vraie à un stage particulier dans l'exécution du protocole. Dans l'analyse du protocole SSL, nous voulons exprimer que le client et le serveur s'authentifient l'un à l'autre. Nous avons donc besoin de quatre variables que nous définissons comme suit :

```
bit PclientInstanceClientServeur = 0;
bit PclientCommetClientServeur = 0;
bit PserveurInstanceClientServeur = 0;
bit PserveurCommetClientServeur = 0;
```

PclientInstanceClientServeur est vraie si et seulement si le client prend part dans une session du protocole avec le serveur.

PserveurInstanceClientServeur est vraie si et seulement si le serveur prend part dans une session du protocole avec le client.

PclientCommetClientServeur est vraie si et seulement si le client termine avec succès une session avec le serveur.

PserveurCommetClientServeur est vraie si et seulement si le serveur termine avec succès une session avec le client.

L'authentification du serveur auprès du client peut alors être vérifiée en s'assurant que *PserveurInstanceClientServeur* doit devenir vraie avant *PclientCommetClient-Serveur*. D'une manière analogue, l'authentification du client auprès du serveur peut

alors être vérifiée en s'assurant que *PclientInstanceClientServeur* doit devenir vraie avant *PserveurCommetClientServeur*.

En notation LTL, cette propriété de précédence peut être exprimée comme suit :

$$(G\ (\ (\ G\ \neg PclientCommetClientServeur)\ \vee\ (\neg PclientCommetClientServeur\ U\ PserveurInstanceClientServeur)\ ))$$
$$\wedge$$
$$(G\ (\ (\ G\ \neg PserveurCommetClientServeur)\ \vee\ (\neg PserveurCommetClientServeur\ U\ PclientInstanceClientServeur)\ ))$$

Les macros *PclientInstance* et *PclientCommet* sont utilisées dans la spécification formelle du protocole SSL pour mettre à jour les variables *PclientInstanceClientServeur* et *PclientCommetClientServeur*. Nous les avons définies en PROMELA de la manière suivante :

Programme 5 *Macros côté client*

```
#define PclientInstance(x, y)
if
:: ((x == client) && (y == serveur)) → PclientInstanceClientServeur = 1
:: else skip
fi
#define PclientCommet(x, y)
if
:: ((x == client) && (y == serveur)) → PclientCommetClientServeur = 1
:: else skip
fi
```

L'appel de la macro *PclientInstance(client, serveur)* est la première instruction exécutée par le processus *Pclient* ; afin de marquer le début de l'instanciation d'une session du protocole par le client. Alors que l'appel *PclientCommet(client, serveur)* est la dernière instruction exécutée par le processus *Pclient* afin de marquer la fin de la session avec le serveur.

Les deux autres macros appelées au même endroit mais dans le processus *Pserveur* sont :

Programme 6 *Macros côté serveur*

```
#define PserveurInstance(x, y)
if
:: ((x == client) && (y == serveur)) → PserveurInstanceClientServeur = 1
```

```
:: else skip
fi
#define PserveurCommet(x, y)
if
:: ((x == client) && (y == serveur)) → PserveurCommetClientServeur = 1
:: else skip
fi
```

Validation de la confidentialité

Le critère de la confidentialité ne peut être validé qu'avec la présence d'un processus intrus en vérifiant que le message échangé entre le client et le serveur ne fait pas partie des connaissances de l'intrus à la fin de la vérification du protocole. Dans le processus *intrus*, on doit déclarer un booléen *message* qui est initialisé à faux. Ce booléen devient vrai si l'intrus arrive à intercepter le contenu du message échangé entre les deux parties impliquées dans la communication. Pour vérifier que la confidentialité du message échangé est respectée à n'importe quel état appartenant à l'espace des états, donc on peut exprimer ce critère simplement par la formule LTL suivante :

$G \neg message$

Validation de l'intégrité

L'intégrité se vérifie en comparant le message envoyé par le client (ou par le serveur) à celui reçu par le serveur (ou par le client).

Résultats de la vérification

L'annexe B expose les résultats de la vérification exhaustive du modèle que nous avons construit pour valider la solution SSL/signature électronique. La vérification des critères de validation exposés plus haut : authentification, intégrité, confidentialité et non répudiation dans tout l'espace des états est positive comme le montre le fichier ssl.spin.ltl.

Aussi les tests par défaut effectués par SPIN et qui sont :

never-claim,

assertion violations,

cycle checks et

invalid endstates

et qui expriment les critères de corrections de la cohérence du protocole en dehors des services de sécurité qu'il fournit, s'avèrent positifs, comme nous le montrons dans l'annexe B.

5.3 Conclusion

L'objectif de la vérification présentée en haut est de valider le protocole {SSL, signature numérique} en l'absence d'un intrus. Cette validation est nécessaire pour toute jointure d'unité de confiance présentée par SecAdvise comme solution à un contexte/transaction. En effet, chaque sélection par SecAdvise constitue un nouveau protocole d'où la nécessité de prouver que ce nouveau protocole est exempt de toute faille intrinsèque.

Cependant, notre validation n'est pas encore complète du fait qu'il manque le modèle de l'intrus. La génération d'un processus intrus par SecAdvise doit, pour des raisons de fiabilité et de qualité de service, être automatique. Ceci revient à dire qu'à partir de la spécification d'un ensemble d'unités, SecAdvise doit être capable de générer d'une manière automatique tout comportement éventuel d'un agent illégitime représenté par un processus intrus.

Dans les chapitres qui suivent, nous faisons le lien entre SecAdvise et les modèles de validation en expliquant théoriquement la procédure à suivre pour valider automatiquement une solution de sécurité. Nous ébauchons aussi un algorithme pour générer automatiquement le processus intrus pour chaque solution.

Chapitre 6

Le modèle de l'intrus

6.1 Connaissances de l'intrus

L'intrus doit interagir avec les participants légitimes d'un contexte/transaction suivant le pouvoir que le modèle de validation de SecAdvise lui fournit. Aussi doit-il être capable de se comporter comme un utilisateur normal du réseau ouvert.

À n'importe quel moment, le comportement de l'intrus dépend des connaissances qu'il a acquises jusqu'à présent. Avant chaque exécution du protocole, il est supposé que l'intrus connaisse un ensemble de données, comme par exemple : l'identité de l'intrus, sa clé publique et privée, l'identité des autres participants, leur clé publique et peut être des clés secrètes qu'il a déjà partagées avec d'autre participants.

À chaque interception de message, l'intrus peut accroître l'ensemble de son savoir. En effet, si le message intercepté est encrypté par une clé dont l'intrus connait la clé inverse, il peut le décrypter et prendre connaissance de son contenu. Si par contre il ne connaît pas la clé inverse, il se contentera de mémoriser tout le message crypté. Cette démarche permet de construire un intrus puissant qui doit extraire le maximum d'informations des messages interceptés.

En plus des messages interceptés et des informations qu'il peut en extraire, l'intrus peut forger d'autres messages à partir de ces derniers. Cette capacité peut être représentée en ajoutant au savoir de l'intrus les données générées à partir de celles interceptées. Cependant, afin de restreindre l'ensemble des messages pouvant être générés par l'intrus, on peut exclure ceux qui ne pourront être acceptés par les participants légitimes du contexte/transaction à sécuriser.

Les participants légitimes et l'intrus sont représentés dans le modèle par des processus PROMELA qui communiquent entre eux via des canaux partagés. Les participants légitimes ne communiquent pas entre eux directement mais tous les messages envoyés sont interceptés par l'intrus qui éventuellement les transfère à la bonne adresse. L'avantage de cette approche qui a été suivie dans plusieurs modèles [14], est de limiter l'espace des états du système et donc la réduction des chemins d'exécution.

L'objectif de SecAdvise est d'automatiser la génération du comportement d'un tel intrus à partir de la spécification formelle des unités de confiance qui rentre dans la composition d'une solution à valider.

En se basant sur la méthode utilisée dans [16] pour générer manuellement le comportement de l'intrus afin de détecter la faille de sécurité dans le protocole *Needham-Shroeder*, on peut réaliser un algorithme de construction automatique du processus intrus. çela demande une analyse préliminaire statique afin d'avoir toutes les informations voulues.

En premier, nous avons besoin de déterminer les ensembles des valeurs possibles que peut prendre chaque variable figurant sur chaque processus du protocole. Cette opération peut être réalisée en effectuant une analyse de flux de données. Les variables dont les processus ne testent pas le contenu peuvent prendre n'importe quelle valeur. Alors que les variables qui sont testées dans des expressions de conditions ne peuvent prendre que les valeurs avec lesquelles on les compare. Prenons l'exemple du chapitre précédent. Si une variable g est utilisée pour prendre comme valeur un identifiant du *serveur* ou *client*, alors les seules valeurs qu'elle peut prendre sont *client*, *serveur* et *intrus*.

Une seconde analyse statique doit être effectuée pour restreindre l'ensemble des connaissances de l'intrus au strict minimum. En tout premier lieu, on a besoin de déterminer l'ensemble des connaissances initiales de l'intrus, qui dans le cas d'un protocole transactionnel, par exemple, est limité à :

- l'ensemble des clés publiques de tous les participants du système,
- la clé privée de l'intrus,
- les certificats de tous les participants du système et
- des données génériques.

Cet ensemble peut voir son contenu augmenter quand l'intrus intercepte des mes-

sages. Si on calcule tous les messages possibles que l'intrus peut intercepter durant l'exécution du protocole, nous pouvons déduire quels sont les messages que l'intrus peut ajouter à son savoir. Par exemple, si l'intrus intercepte le message : {*m_client, client*}*PK(intrus)*, il peut connaître le message *m_client*.

Pour éviter la redondance d'éléments appris, nous supposons que l'intrus sauvegarde l'élément appris sous sa forme la plus élémentaire. Par exemple, si le message : *m_client, clientPK(intrus)* est intercepté, nous supposons alors que le processus intrus sauvegarde *m_message*, et non tout le message : *m_client, clientPK(intrus)*, puisque ce dernier peut être construit à partir de *m_client* et de l'identifiant *client*, qui font déjà partis des connaissances de l'intrus. En d'autre mots, le processus intrus sauvegarde le message en sa forme la plus complexe seulement s'il ne peut pas le décrypter. Par exemple, si l'intrus intercepte le message *m-client, clientPK(serveur)*, il ne peut que sauvegarder le message en entier puisqu'il ne connaît pas la clé privée du serveur.

6.2 Génération automatique de l'intrus

Par convention, les éléments appris par l'intrus au cours de l'exécution du protocole sont représenté dans le processus intrus par des variables booléennes locales. De plus, pour des raisons de génération automatique de code, on donne à ces variables le même nom que la donnée apprise avec un préfixe « *k* ». Par exemple, la variable *k_m_client* représente la connaissance par l'intrus du nonce *m_client*. Similairement, la connaissance par l'intrus d'une donnée structurée peut être représentée par une variable de type bit dont le nom comprend le préfixe « *k* » et les champs dans l'ordre de leur occurrence.
Par exemple, la variable *k_m_client_client_serveur* va représenter la connaissance du message *m_client, clientPK(serveur)*.

Comme l'ensemble des messages que l'intrus peut intercepter est un ensemble fini, nous aurons aussi un ensemble fini de messages pouvant être ajoutés à l'ensemble de connaissances de l'intrus.

Appelons *RCEIVED* l'ensemble de tous les messages que l'intrus peut intercepter, et pour chaque message m appartenant à *RECEIVED*, appelons $LEARNED_m$ l'ensemble des éléments qu'il peut apprendre du message m.

Une autre restriction de l'ensemble de connaissances de l'intrus est possible si on exclut les messages, qui même s'ils faisaient partie du bagage de connaissances de l'intrus, ne pourraient jamais être utilisés par l'intrus pour générer des messages valides. Si par exemple l'intrus prend connaissance du message *m_serveurPK(client)*, il peut potentiellement l'envoyer à *serveur* mais ce dernier ne pourra pas l'accepter, parce qu'il ne respecte aucun format de message du protocole. Il n'est donc pas utile à l'intrus de connaître ce message.

L'ensemble des données potentiellement utiles à l'intrus peut être calculé en listant tous les messages valides que l'intrus peut éventuellement envoyer aux autres principaux. Appelons cet ensemble *UTIL*.

On doit déterminer ensuite pour chacun de ces messages, quelles données l'intrus peut utiliser pour construire de tels messages. Appelons $NeedKnowledge_m$ l'ensemble des éléments dont l'intrus a besoin pour construire un message m qui pourra lui être utile, avec $m \in UTIL$.

L'ensemble des éléments que l'intrus peut connaître et qui seront utiles dans sa tâche d'intrusion est donc défini comme suit :

$$VarUtil = \bigcup_{m \in RECEIVED} LEARNED_m \cap \bigcup_{m \in UTIL} NeedKnowledge_m$$

Le processus de type intrus peut être construit de la manière suivante :

Programme 7 *Processus Intrus*

```
proctype ProcessusIntrus() {
    /*la connaissance initiale de l'intrus, des constantes
    */
    Les identifiants
    Les clés publiques des participants et de l'intrus
    La clé privée de l'intrus
    des variables génériques
    des variables de services
    /*l'ensemble des variables qui vont contenir les valeurs que l'intrus va connaître.
    ces variables sont représentées par les noms des éléments de l'ensemble VarUtil
    */
    bit k_nom_variable = 0
    ...
    do
    :: écriture ou lecture d'un canal
    :: ...
    od
}
```

Dans la partie déclaration de variables, nous avons une variable de type bit pour chaque élément de l'ensemble $VarUtil$. Ces variables sont initialisées à 0, parce qu'initialement la valeur de la donnée correspondante n'est pas connue de l'intrus. Le processus leur donnera la valeur 1 s'il arrive à intercepter cette valeur au cours de l'exécution du protocole.

La partie du processus *ProcessusIntrus* représentant le comportement de l'intrus est une boucle do infinie dans laquelle le processus envoie sans cesse des messages sur les canaux partagés par les processus. Comme les opérations sont atomiques, l'état du processus intrus est déterminé par la valeur courante de ses variables dont le nom commence par « k » comme il a été mentionné plus haut. Chaque branchement dans la boucle de répétition représente un input ou un output dans les canaux globaux. Pour chaque message envoyé par l'intrus, il y a un branchement de type output. Certains de ces branchements ont une pré-condition qui les active, alors que d'autres branchements ne sont pas conditionnés par l'ensemble des connaissances de l'intrus. Par exemple, le branchement :

:: canal ? serveur, var_generic, client, serveur

représentant l'envoi par l'intrus du message $\{var\_generic, client\}PK(serveur)$ au serveur, est toujours exécutable si un processus, dans ce cas le processus associé au participant serveur, est prêt à être synchronisé, avec le processus intrus.
Par contre, le branchement :

:: canal ? (kNserveur → serveur : X), Nserveur, Nserveur, serveur

représente l'envoi par l'intrus du message $\{Nserveur, Nserveur\}PK(serveur)$ à *serveur*. L'expression conditionnelle se trouvant à la première position du branchement est égale à *serveur* si l'intrus connaissait *Nserveur*, sinon elle est égale à une identité inexistante X. Cela revient à dire que l'intrus peut envoyer le message *Nserveur*, $NserveurPK(serveur)$ si et seulement si il connaît *Nserveur*. Aussi, dans la boucle principale du code de l'intrus trouve-t-on un branchement servant à lire de chaque canal. La lecture des canaux sert à enregistrer le contenu du message dans les variables de service déclarées plus haut.

La mise à jour des variables de connaissance déclarées plus haut peut se faire par des macros, qui automatiquement ignorent les éléments appris et qui ne sont d'aucune

utilité à la fonction d'intrusion.

La macro suivante a le rôle de mettre à jour les deux variables *kNserveur* et *kNclient* :

```
#define k1(x1)
if
::(x1 == Nserveur) → kNserveur = 1
::(x1 == Nclient) → kNclient = 1
:: else skip
fi;
```

Alors que la macro k2 a pour rôle de mettre à jour la connaissance de la clé secrète du serveur :

```
#define k2(x1, x2)
if
:: (x1 == Nserveur && x2 == serveur) → k_Nserveur_serveur = 1
else skip
fi
```

Si l'intrus reçoit le message $\{Nserveur, serveur\}PK\{I\}$ du canal can1, la variable k_N_serveur est remise à 1, et, s'il reçoit le message $\{Nclient\}PK\{client\}$ du canal can2, la variable *k_Nclient_client* est remise à 1.

6.3 Synthèse

Dans les chapitres précédents, nous avons abordé la validation comme critère de sélection de solutions de confiance par SecAdvise. Ce dernier doit, en effet, s'assurer qu'un ensemble d'unités de confiance couvre effectivement l'ensemble de risques que les unités prétendent couvrir.

La méthode que nous avons proposée dans les chapitres 4 et 5 consiste à utiliser l'outil de spécification formelle PROMELA, jumelé avec le paquetage SPIN pour la vérification automatique de l'espace d'états du système ainsi décrit. Afin de généraliser la méthode de vérification à n'importe quelle solution de sécurité et en se basant sur la méthode de vérification de l'étude de cas, nous ajoutons à SecAdvise les notations suivantes :

EFSM l'ensemble des machines à états finis étendues représentants le fonctionnement des unités de confiances de l'ensemble U. Un tel ensemble fait le lien entre une unité de confiance u et sa représentation sous forme d'automate

étendu. D'une part, il sert à stocker d'une manière formelle la spécification d'une unité de confiance et d'autre part à automatiser le passage spécification formelle/implémentation comme nous l'avons illustré dans l'exemple à la section 4.5, qui consistait à montrer l'équivalence entre une implémentation en langage PROMELA et un EFSM.

$efsm_u \in \mathbf{EFSM}$ est un EFSM représentant le fonctionnement de l'unité de confiance u. On a alors $efsm_u \in \mathbf{EFSM}$ et $efsm_{\mathbf{U}} = \bigcup_{u \in \mathbf{U}} efsm_u$ tel que $U \in \mathcal{P}(\mathbf{U})$;

$\mathbf{D}$ l'ensemble des prédicats associés aux risques de l'ensemble $\mathbf{R}$

Pour chaque risque, on associe un prédicat de vérification. Ce prédicat peut être une formule LTL ou tout autre formule de vérification. Il permet de vérifier si le risque de sécurité est couvert par une ou plusieurs unités de confiance. Le prédicat est indépendant des unités de confiances couvrant le risque pour lequel est associé le prédicat en question. Autrement dit, le prédicat à vérifier est le même pour n'importe quel ensemble d'unités couvrant le risque en question ;

d_r un prédicat associé au risque r. On peut avoir plusieurs prédicats associés à un risque donné ;

$\mathbf{D}_r$ l'ensemble de tous les prédicats associés au risque r ;

$\mathbf{D}_{\mathbf{R}_c}$ l'ensemble des prédicats associés aux risques de l'ensemble $\mathbf{R}_c$ (l'ensemble des risques à couvrir dans le contexte/transaction c) ;

$\mathbf{I}$ l'ensemble de comportements d'intrus décrits au moyen d'EFSMs $\mathbf{I} \in \mathcal{P}(\mathbf{EFSM})$;

$i_{\mathbf{U}}$ l'EFSM décrivant le comportement de l'intrus servant à la validation de l'ensemble d'unités de confiance $\mathbf{U}$. La génération automatique de l'intrus est décrite dans le chapitre précédent. $i_{\mathbf{U}} \in \mathbf{I}$;

En se basant sur les éléments nouvellement introduits au modèle théorique de SecAdvise, on peut formuler la phase de validation que SecAdvise doit entreprendre pour proposer une solution de confiance couvrant les risques d'un contexte/transaction c en deux étapes :

1. la vérification de chaque prédicat de l'ensemble $\mathbf{D}_{\mathbf{R}_c}$ dans chaque état de l'automate $efsm_{\tilde{U}_c}$ et

2. la vérification de chaque prédicat de l'ensemble $\mathbf{D}_{\mathbf{R}_c}$ dans chaque état de l'automate $efsm_{\tilde{U}_c \cup i_{\tilde{U}_c}}$

La première validation consiste à vérifier si un risque de sécurité $r \in R_c$ n'est pas couvert par la solution de confiance proposée par SecAdvise $\tilde{U}_c$. Alors que la deuxième vérification consiste à valider la couverture des risques R_c par $\tilde{U}_c$ en la présence d'un intrus.

6.4 Conclusion

Dans ce chapitre nous avons décrit d'une manière informelle la génération automatique du comportement d'un intrus pour que l'aviseur SecAdvise puisse valider une solution de confiance donnée. Le comportement d'un tel intrus doit tenir compte de la structure des processus représentant le comportement des participants légitimes. De plus, SecAdvise doit générer automatiquement l'ensemble de connaissances initiales de l'intrus. Cet ensemble doit s'agrandir au fur et à mesure que l'intrus intercepte et décrypte les messages échangés entre les différents participants à l'aide de macros de mises à jour des variables de connaissance.

Chapitre 7

Conclusion

L'automatisation et la dématérialisation des échanges monétaires ne sont pas des objectifs en soi, mais uniquement des moyens pour améliorer la productivité des institutions financières. À cet égard, la surabondance de normes, de standards et de solutions pour assurer la sécurité des échanges et parfois leur concurrence peut freiner le développement du commerce électronique.

Dans ce mémoire, nous avons rappelé le rôle rempli par SecAdvise et avons ajouté d'autres critères de sélection à l'aviseur SecAdvise, à savoir la diminution des risques sous des contraintes et avec un moindre coût. Aussi, a-t-on ajouté la notion de validation qui permet de s'assurer que la solution proposée par SecAdvise est exempte de toute faille intrinsèque. Ce dernier critère est très important dans la sélection des meilleurs mécanismes ou systèmes de sécurité couvrant les risques qui menacent les systèmes du commerce électronique.

La conception de SecAdvise ne sera complète qu'avec la réalisation des travaux futurs suivants :

- *conception d'un algorithme pour la génération automatique du comportement de l'intrus à partir de la spécification des unités de confiance d'une solution couvrant un ensemble de risques.*

  À partir du pseudocode présenté au chapitre 6, on peut élaborer une définition plus complète d'un algorithme de génération automatique du comportement de l'intrus. Cet algorithme aura comme paramètres d'entrée une spécification formelle de l'union des unités de confiance constituant la solution à valider ;
- *validation de cet algorithme, ce qui revient à trouver une méthode pour prouver*

que le comportement de l'intrus généré est le strict minimum voulu pour valider une solution fonctionnant dans un milieu ouvert.

Cette validation est nécessaire pour s'assurer que l'algorithme élaboré génère exactement l'ensemble des messages dont l'intrus aura besoin dans sa tâche d'intrusion, ni plus ni moins ;

- *conception formelle d'un patron de protocole pour SevAdvise en décrivant le format des messages qui doivent être échangés entre l'aviseur et les participants, en décrivant l'automate du déroulement des échanges du coté client (les participants appartenant à l'ensemble* $\mathbf{P}_c$*) et du coté du serveur de SecAdvise.*

  En effet, Il n'y a eu jusqu'à présent qu'une conception théorique de SecAdvise, il faudra mettre en oeuvre cette conception en commençant par donner une spécification formelle du protocole SecAdvise. Ce dernier sera un protocole distribué entre un serveur SecAdvise et plusieurs autres participants voulant sécuriser leur communication. Cette spécification doit inclure une description :
  - des services à être fournis par SecAdvise,
  - des hypothèses sur l'environnement dans lequel SecAdvise sera exécuté,
  - du format et de l'encodage de chaque message échangé entre SecAdvise et les participants,
  - des règles de procédure régissant le protocole SecAdvise.
- *validation de la spécification.*

  On peut valider la spécification par un des outils existants de conception et de validation des protocoles distribués comme PROMELA, SDL, ESTEL, etc ;
- *implémentation de SecAdvise.*
- *vérification par des tests fonctionnels de l'aviseur implémenté.* Les tests fonctionnels consistent à vérifier la coïncidence entre la spécification formelle et l'implémentation réelle du protocole.

La phase la plus importante est celle de la spécification formelle du protocole SecAdvise et sa validation du fait que l'implémentation ne pose généralement pas problème si la spécification est validée.

Dans ce mémoire, nous avons poussé l'étude sur le modèle théorique de l'aviseur de sécurité SecAdvice et nous avons présenté par un exemple l'idée derrière la valida-

tion automatique d'une union de mécanismes de sécurité. Une fois conçu et validé, SecAdvice répondra à plusieurs demandes :

- minimisation des risques menaçant les transactions dans les systèmes ouverts,
- minimisation du coût de la couverture de ces risques sous des contraintes d'environnement d'exécution,
- validation automatique de solutions de confiance en tenant compte de participants illégitimes et de l'union de mécanismes de sécurité couvrant l'ensemble des risques menaçant les opérations électroniques.

Bibliographie

[1] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on software engineering*, 22(1) :6–15, 1996.

[2] G. Calvanese, D. De Giacomo and M.Y. Vardi. Reasoning about Actions and Planning in LTL Action Theories. *KR'02*, 2002.

[3] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature : Version 1.0. A continually updated library of protocols analysed in the literature, available at www.cs.york.ac.uk/ jac, Novembre 1997.

[4] M. Curphey and D. Endler. A Guide to Building Secure Web Applications. Disponible à http ://www.owasp.org, 2002.

[5] M.H. Deitel, J.P. Deitel, B. Duwaldt, and L.K. Trees. *Web Services : A Technical Introduction.* Prentice Hall PTR, first edition, 2002.

[6] E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. In *Communications of the ACM*, volume 18(8), pages 453–457, 1975.

[7] F.J.T. Fábrega, J.C. Herzog, and J.D. Guttman. Strand Spaces : Proving Security Protocols Correct. *Journal of Computer Security*, 1999.

[8] A.O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. Disponible à http ://wp.netscape.com/eng/ssl3/draft302.txt, 1996.

[9] S. L. Garfinkel. *PGP : Pretty Good Privacy.* O'Reilly and Associates, 1995.

[10] V. Hassler. *Security Fundamentals for E-commerce.* Computer Security Series. Artech House, 2001.

[11] G.J. Holzmann. *Design and Verification of Computer Protocols.* Prentice hall, Englewood Cliffs, NJ, international edition edition, November 1990.

[12] IETF/RFC. *PGP message exchange formats*, 1996.

[13] ISO/IEC StandardInternational Organisation for Standardization. *Systèmes de traitement de l'information - Interconnexion des systèmes ouverts - Modèle de référence de base*, number 7498-2, 1989.

[14] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. In *Information Processing letters*, volume 56(3), pages 131–133, 1996.

[15] G. Lowe. Breaking and fixing the Needham-Shroeder public-key protocol using FDR. In *Proceeding of TACAS96*, pages 147–153, LNCS 1055, Springer-Verlag, 1996.

[16] P. Maggi and R. Sisto. Using Spin to Verify Security Properties of Cryptographic Protocols. *Lecture Notes in Computer Science, Springer-Verlag Heidelberg*, 2002.

[17] J Massey and X. Lai. A proposal for a new block encryption standard. *Proc. Eurocrypt'91*, 1991.

[18] R. Rivest. *The MD5 Message-Digest Algorithm.* MIT Laboratory for Computer Science and RSA Data Security, Inc., 1992. Disponible à http ://www.faqs.org/rfcs/rfc1321.html.

[19] R.L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[20] S. Robles, S. Poslad, J. Borrell, and J. Bigham. A Practical Trust Model for Agent-Oriented Electronic Business Applications. In *Proc. of the 4th Int'l Conf. on Electronic Commerce Research (ICECR-4)*, volume 2, pages 397–406, Dallas, Texas, USA, November 2001.

[21] R. Saliba. SecAdvise : un aviseur de mécanismes de sécurité. Master's thesis, Université de Montréal Faculté des Arts et des Sciences Département d'Informatique et de Recherche Opérationnelle, Avril 2002.

[22] R. Saliba, G. Babin, and P. Kropf. SecAdvice : A Security Mechanism Advisor. *Distributed Comunities on the Web (DCW 2002)*, pages 35–40, Avril 2002.

[23] M.H. Sherif and A. Serhrouchni. *La monnaie électronique — système de paiement sécurisé.* Eyrolles, 2000.

[24] Union Internationale des Télécommunications. *Recommandation X.800. Architecture de sécurité pour l'interconnexion en systèmes ouverts d'application du CCITT*, number version 3, 1991.

[25] Union Internationale des Télécommunications. *Recommandation X.509. Technologies de l'information - Interconnexion des systèmes ouverts - Cadres de sécurité pour systèmes ouverts : cadre d'authentification*, number version 3, 1996.

[26] Visa and MasterCard. SET Secure Electronic Transaction Specification Book 3 version 1.0 : Formal Protocol Definition, 1997. Disponible à http ://www.setco.org/download/set_bk3.pdf.

Annexe A

Specification de SSL

Programme 8 *Processus client et serveur*

```
chan HelloRequest = [0] of {mtype};
chan ClientHello = [0] of {mtype, mtype};
chan ServerHello = [0] of {mtype, mtype};
chan Certificate = [0] of {mtype, mtype mtype, mtype};
chan CertificateRequest = [0] of {mtype, mtype};
chan ServerHelloDone = [0] of {mtype, mtype};
chan ClientKeyExchange = [0] of {mtype, mtype};
chan CertificateVerify = [0] of {mtype, mtype};
chan Finished = [0] of {mtype, mtype};
chan NoCertificate = [0] ofof {mtype};
chan ChangeCipherSpec = [0] of {mtype};
mtype = { client, serveur, intrus,
hello_request, client_hello,
server_hello, certificat_client, certificat_serveur,
certificate_request, server_hello_done, pre_master_secret,
handshake_messages_hash,finished_hash,
handshake_message_master_secret, gD, R}
/* Formats des messages du protocole SSL

    HelloRequest{serveur}

    ClientHello{client, client_hello}

    ServerHello{serveur, server_hello}

    Certificate{client (ou serveur), certificat_client ou

    (certificat_serveur), {certificat_client}KS{client} (ou

    {certificat_serveur}KS{serveur}) }

    CertificateRequest{serveur, certificate_request}

    ServerHelloDone{serveur, server_hello_done}
```

```
    ClientKeyExchange{client, {pre_master_secret}KP{serveur}}
    CertificateVerify{serveur, {{handshake_messages}HASH}PS{client} }
    Finished {client (ou serveur), {handshake_messages, client (ou serveur),
    masterSecret}HASH},
    NoCertificate{client}
*/
proctype Pserveur (
mtype self;
mtype party;
mtype server_hello1;
mtype certificat_serveur1;
mtype  server_hello_done1;
mtype  certificate_request1;
mtype  server_hello_done2;
mtype  handshake_messages_hash1;
mtype  handshake_message_master_secret1 ) {
    mtype v1;
    /*client_hello
    */
    mtype v2;
    /*pre_master_secret
    */
    mtype v3;
    mtype v4;
    /* certificat_client1
    */
    mtype v5;
    /*certificat_client2
    */
    mtype v6;
    mtype v7;
    /*finished_hash
    */
    mtype v8;
    atomic {
        HelloRequest ? party;
    }
    atomic {
        ClientHello ! eval(self), v1;
    }
    atomic {
        ServerHello ? party, server_hello1;
```

```
}
atomic {
    Certificate ? party,certificat_serveur1 ,certificat_serveur1, self;
}
if
:: atomic {
    ServerHelloDone ?party,server_hello_done1;
}
atomic  {
    ClientKeyExchange !eval(self), v2, v3
}
:: atomic  {
    CertificateRequest ?party,certificate_request1;
}
atomic {
    ServerHelloDone ?party, server_hello_done2;
}
if
::  atomic {
    Certificate ! eval(self), v4, v5, v6;
}
atomic {
    ClientKeyExchange ! eval(self), v2, v8;
}
atomic {
    CertificateVerify ? party,handshake_messages_hash1, party
}
::  atomic {
    NoCertificate ! eval(self);
}
atomic {
    ClientKeyExchange ! eval(self), v2, v8
}
fi
fi;
atomic {
    ChangeCipherSpec ! eval(self);
}
```

```
    atomic {
        Finished ! eval(self), v7;
    }
    atomic {
        ChangeCipherSpec ? party;
    }
    atomic {
        Finished ? party, handshake_message_master_secret1;
    }
}
proctype Pclient (
mtype self;
mtype party;
mtype client_hello1;
mtype pre_master_secret1;
mtype certificat_client1;
mtype finished_hash1 ) {
    mtypev1 ,
    /*server_hello
    */
    v2,
    /*certificat_serveur
    */
    v3,
    /*certificat_serveur
    */
    v4,
    /*serveur
    */
    v5,
    /*server_hello_done
    */
    v6,
    /*certificate_request
    */
    v7,
    /*server_hello_done
    */
    v8,
    /*handshake_messages_hash
    */
    v9,
    /*client
```

```
*/
v10;
/*{handshake_messages, client (ou serveur), masterSecret}HASH}
*/
atomic{
    HelloRequest !eval(self);
}
atomic{
    ClientHello ?party, client_hello1;
}
atomic{
    ServerHello !eval(self), v1;
}
atomic{
    Certificate !eval(self), v2, v3, v4;
}
if
::atomic{
    ServerHelloDone !eval(self), v5;
}
atomic{
    ClientKeyExchange ?party, pre_master_secret1, party
}
::atomic{
    CertificateRequest !eval(self), v6;
}
atomic{
    ServerHelloDone !eval(self), v7;
}
if
::atomic{
    Certificate ?party, certificat_client1, certificat_client1, self;
}
atomic{
    ClientKeyExchange ?party, pre_master_secret1, party;
}
atomic{
    CertificateVerify !eval(self), v8, v9
```

```
    }
    ::atomic{
        NoCertificate ?party;
    }
    atomic{
        ClientKeyExchange ?party, pre_master_secret1, party
    }
    fi
    fi;
    atomic{
        ChangeCipherSpec ?party;
    }
    atomic{
        Finished ?party, finished_hash1;
    }
    atomic{
        ChangeCipherSpec !eval(self);
    }
    atomic{
        Finished !eval(self), v10;
    }
}
init
{
    atomic {
        run Pserveur(serveur,
        client,
        server_hello,
        certificat_serveur,
        server_hello_done,
        certificate_request,
        server_hello_done,
        handshake_messages_hash,
        handshake_message_master_secret);
        run Pclient(client,
        serveur,
        client_hello,
        pre_master_secret,
        certificat_client,
        finished_hash );
```

```
    }
}
```

Annexe B

Résultats

```
(Spin Version 4.0.4 -- 08 April 2003)
        + Partial Order Reduction

Full statespace search for:
        never-claim
        assertion violations
        cycle checks
        invalid endstates

State-vector 64 byte, depth reached 24, errors: 0
     191 states, stored
     137 states, matched
     328 transitions (= stored+matched)
     835 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

2.622   memory usage (Mbyte)

unreached in proctype Pclient
        (0 of 18 states)
unreached in proctype Pserveur
        (0 of 18 states)
unreached in proctype :init:
        (0 of 8 states)
```

```
0.02user 0.02system 0:00.12elapsed 32%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (131major+1210minor)pagefaults 0swaps


/* contenu du fichier ssl.spin.ltl */

#define p       PclientCommetClientServeur
#define q       PserveurInstanceClientServeur

        /*
         * Formula As Typed: [] ( ([]  !(p))  ||  (!(p) U (q)) )
         * The Never Claim Below Corresponds
         * To The Negated Formula !([] ( ([]  !(p))  ||  (!(p) U (q)) ))
         * (formalizing violations of the original)
         */

never {    /* !([] ( ([]  !(p))  ||  (!(p) U (q)) )) */
T0_init:
        if
        :: (! ((q)) && (p)) -> goto accept_S11
        :: (! ((q)) && (p)) -> goto accept_all
        :: (! ((q))) -> goto T0_S14
        :: (! ((q)) && (p)) -> goto accept_S2
        :: (1) -> goto T0_init
        fi;
accept_S11:
        if
        :: (! ((q))) -> goto accept_S11
        :: (! ((q)) && (p)) -> goto accept_all
        fi;
accept_S2:
        if
        :: ((p)) -> goto accept_all
```

```
        :: (1) -> goto T0_S2
        fi;
T0_S14:
        if
        :: (! ((q)) && (p)) -> goto accept_S11
        :: (! ((q)) && (p)) -> goto accept_all
        :: (! ((q))) -> goto T0_S14
        :: (! ((q)) && (p)) -> goto accept_S2
        fi;
T0_S2:
        if
        :: ((p)) -> goto accept_all
        :: (1) -> goto T0_S2
        fi;
accept_all:
        skip
}

#ifdef NOTES
q precedes p


#endif
#ifdef RESULT
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.0.4 -- 12 April 2003)
        + Partial Order Reduction

Full statespace search for:
        never-claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid endstates       - (disabled by never-claim)
```

```
State-vector 68 byte, depth reached 30, errors: 0
     379 states, stored
     813 states, matched
    1192 transitions (= stored+matched)
    2321 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

2.622   memory usage (Mbyte)

unreached in proctype PIni
        (0 of 18 states)
unreached in proctype PRes
        (0 of 18 states)
unreached in proctype PI
        line 185, state 128, "-end-"
        (28 of 128 states)
unreached in proctype :init:
        (0 of 8 states)
0.05user 0.01system 0:00.07elapsed 83%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (133major+1214minor)pagefaults 0swaps

#endif
```