Mémoire de fin d'études

En vue de l'obtention du Grade de Master 2 Recherche
en Informatique.

Spécialité : FOndement des **L**ogiciels **S**écurisés.

Génération de tests de protocoles cryptographiques par mutations de modèles

Présenté par :

M. Rafik KHEDDAM

Soutenu le 18 juin 2010, devant le Jury :

- **M. Jacques Julliand**, Professeur a l'université de Franche-Comté, (***examinateur***).
- **M. Frédéric Dadeau**, Maître de conférences a l'université de Franche-Comté, (***encadrant***).
- **M. Pierre-Cyrille Héam**, Maître de conférences HDR a l'université de Franche-Comté, (***encadrant***).

-- Année 2009/2010 --

Remerciements

Je remercie en premier lieu mes promoteurs, Monsieur Frédéric Dadeau et Monsieur Pierre-Cyrille Héam, pour leur disponibilité et tout le temps qu'ils m'ont consacré. Je les remercie aussi pour tous les conseils et les orientations qu'ils n'ont cessées de me donner durant toute au long de mon stage.

Mes remerciements s'adressent également à l'ensemble des enseignants du Département D'informatique de l'UFR Sciences et Techniques de l'université de Franche-Comté pour tous leurs efforts afin de nous assurer une meilleure formation.

Enfin, je remercie vivement tous ceux qui m'ont aidé, de près ou de loin, à la concrétisation de ce projet.

Rafik Kheddam.

Table des matières

III. Contributions

IV. Conclusion et perspectives

V. Annexes

VI. Bibliographie

Liste des figures

I. Introduction

Introduction et problématique

Depuis l'avènement d'Internet, les réseaux informatiques prennent de plus en plus de place dans les sociétés modernes et dans notre vie en général. En quelques années, l'informatique s'est imposée et ses utilisateurs ont maintenant plus de besoins et font appel de plus en plus à ces réseaux informatiques dans tous les domaines d'activité.

La pression du marché qui impose des temps de développement de logiciel plus courts et des coûts plus faibles a fait que la démocratisation de l'informatique se fait rapidement. Mais en contre partie, ça a engendré une explosion de la criminalité informatique notamment des intrusions dans les réseaux de ces sociétés ; conséquence directe de leurs ouvertures vers l'extérieur (*vers leurs partenaires*) et de la rapidité du développement des applications utilisées pour leurs communications.

Ces attaques exploitent des failles trouvées dans les différents systèmes qui composent les réseaux et coûtent lourdement à ces entreprises tant en argent qu'en temps, ce qui a fait sentir le besoin de vérifier et de valider la sûreté de l'ensemble du code du logiciel et spécialement dans le domaine du transport des données. De ce fait, de nombreuses solutions de protection des données ont vu le jour notamment les protocoles cryptographiques (qui seront l'objet de notre étude) et qui doivent donc être testés et validés avant leurs mises en marche. Dans notre cas, on s'intéressera à une technique de test bien particulière, qui est apparue récemment, mais qui a des possibilités et un avenir promettant. Ce procédé est connu sous l'appellation du ***test par mutation,*** que nous allons développer un peu plus loin dans ce mémoire.

Contexte du stage

Au sein du Laboratoire d'Informatique de l'université de Franche-Comté (***LIFC***) une partie de l'équipe ***VESONTIO*** [1] travaille dans le cadre d'un projet Européen nommé ***SecureChange***, sur la validation de la sécurité de systèmes embarqués, par le biais de test générés à partir de modèles formels.

Dans le cadre de l'activité de test à partir de modèles qui concerne le LIFC, nous nous somme intéressés pour ce projet de Master, à la génération de tests à partir de mutations de modèles. Le principe est de partir d'un modèle décrivant le fonctionnement nominal d'un système et d'y introduire volontairement des erreurs, créant ainsi des mutants. Le processus de génération de tests utilise les mutants produits et cherche ainsi à produire des jeux de tests permettant de les détecter.

Nous avons appliqué ces principes à des modèles formalisant des protocoles de sécurité. Ainsi, nous avons proposé un modèle de fautes pertinent applicable sur des protocoles cryptographiques, que nous avons développé et implémenté particulièrement à l'aide de l'outil javaCC, ainsi que les outils d'analyse et de validation de ces protocoles pour la génération des tests, comme ***AVISPA*** [2] auquel une partie de l'équipe a participé il y a quelques années.

Plan du mémoire

I. Introduction

Cette partie, comme vous l'avez vue, donne une présentation générale du mémoire et expose la problématique et les justificatifs de notre démarche.

II. Etat de l'art

Ce chapitre donne une vue générale sur le principe du test par mutation et les protocoles cryptographiques, ainsi qu'une vue de l'ensemble des outils que nous avons utilisés dans ce projet, tel que ***AVISPA*** et ***javaCC*** (***Java C****ompiler* ***C****ompiler*) qui est un générateur d'analyseurs lexico-syntaxiques.

III. Contributions

Ce chapitre présente et explique notre modèle de fautes c'est-à-dire les mutations proprement dites, leurs implémentations avec le langage JAVA, et enfin l'expérimentation de ce modèle pour la génération des cas de tests pertinents qui est le but final de ce projet.

IV. Conclusion et perspectives

Pour clore ce mémoire, nous concluons par un résumé ce projet, et nous présentons les extensions possibles de notre modèle de fautes.

[1] **VE**rification, **S**pécificati**ON**, Test et **I**ngénierie des m**O**dèles

[2] **A**utomated **V**alidation of **I**nternet **S**ecurity **P**rotocols and **A**pplications

II. Etat de l'art

Introduction

La généralisation des services sur Internet pousse les professionnels (les entreprises) ou les particuliers à effectuer de plus en plus d'opérations confidentielles à distance : vente en ligne ou commerce électronique, échanges de données sensibles, gestion des comptes bancaires, ce qui engendre un besoin crucial de protection de ces données-là.

Afin de préserver la confidentialité et l'intégrité de ces échanges (transactions), de nombreuses solutions logicielles ont été mises en place notamment les protocoles de sécurité. Le manque de temps rend actuellement difficile ("voire impossible") d'assurer une analyse et une validation rigoureuses de ces derniers, ce qui engendre des failles dans ces systèmes qu'un Intrus peut exploiter pour avoir accès à des données qui ne lui sont pas destinées.

Dans ce cadre, et afin de limiter ces risques, un projet européen AVISPA [10] qui a réuni des partenaires académiques (équipe CASSIS du LORIA et de l'INRIA Lorraine, DIST Université de Gènes, ETH Zürich) et industriel (Siemens de Munich) a vu le jour en 2003 et a abouti au lancement du logiciel éponyme : un outil d'analyse de protocoles cryptographiques. Cet outil, nous le présenterons un peu plus loin dans ce chapitre, mais avant, nous donnerons quelques rappels sur la cryptographie.

1. Cryptographie

Depuis sa création, le réseau Internet a tellement évolué qu'il est devenu un outil essentiel de communication mais qui présente tout de même des risques. Les transactions faites à travers le réseau peuvent être interceptées, d'autant plus que les lois juridiques ont du mal à se mettre en place sur Internet, il faut donc garantir la sécurité de ces informations, c'est la cryptographie qui s'en charge.

La cryptographie [06] était utilisée depuis bien longtemps ; à l'époque romaine, lorsque Jules César envoyait des messages à ses généraux, il ne faisait pas confiance à ses messagers. Aussi remplaçait-il chaque A dans ses messages par un D, chaque B par un E, et ainsi de suite à travers l'alphabet. Seul quelqu'un qui connaissait la règle « décalé de 3 » pouvait déchiffrer ses messages. Aujourd'hui, la cryptographie est utilisée dans divers applications réseaux telles que la messagerie électronique, les réseaux privés virtuels...

La cryptographie [05] désigne l'ensemble des méthodes ou techniques (*chiffrement, signature numérique et certificat*) permettant de garantir intégrité, authenticité, confidentialité des informations sensibles, et on appelle la personne qui la pratique un *cryptographe*.

- **La cryptanalyse :**

  C'est l'étude des procédés cryptographiques dans le but de trouver des faiblesses, et en particulier de pouvoir décrypter des textes chiffrés. Le ***décryptement*** est l'action qui consiste à trouver le message en clair sans connaître la clef de déchiffrement, et on appelle la personne qui pratique cet art un *Cryptanalyste*.

On appelle la science qui englobe la cryptographie et la cryptanalyse, ***la cryptologie***

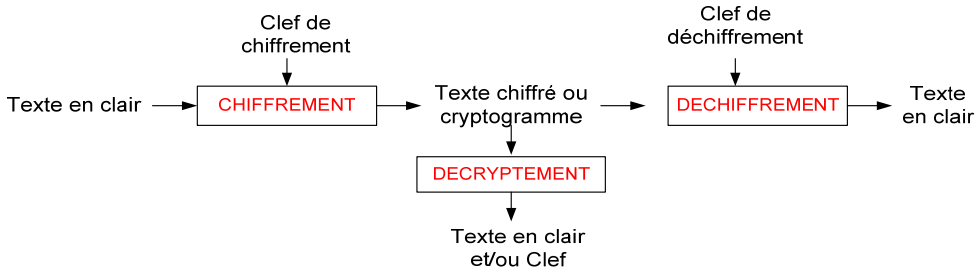


Figure 1.1 : Schéma représentant la cryptologie

<u>La cryptographie permet d'assurer :</u>

1.1. La confidentialité

La confidentialité consiste à assurer que seules les personnes autorisées à manipuler des ressources données aient accès, donc les rendre inintelligibles et inaccessibles par d'autres personnes qui ne possèdent pas ce privilège. Cette propriété est assurée par le procédé du chiffrement.

1.1.1. Le chiffrement

Pour crypter un message ou un texte qu'on appellera ***texte en clair***, on lui applique une série d'opérations simples telles que la substitution et la permutation suivant des règles bien définies qui ne sont connues que par l'émetteur et le récepteur du message (dans le cas le cas d'un chiffrement symétrique) dans le but de

le rendre inintelligible **pour les tiers non autorisés** (**cryptogramme** ou **texte chiffré**) et on appelle ce procédé ***chiffrement*** [01].

Inversement, le **déchiffrement** est l'action qui permet de reconstruire le texte en clair à partir du texte chiffré. Dans le monde de l'informatique moderne, les transformations en question sont des algorithmes construits à base de fonctions mathématiques qui dépendent d'un paramètre qu'on appelle clé de chiffrement/déchiffrement.

- **Clé :** Ensemble des données d'entrée de l'algorithme qui transforme le texte clair en texte chiffrée et inversement. On notera la clé privé d'un agent A par Ka. La clé partagée entre A et B par Kab et un message M chiffré par une clé k est représenté par $\{M\}_K$.

Il existe deux grandes familles d'algorithmes cryptographiques à base de clefs :

a. <u>Les algorithmes à clef privée :</u> (*Chiffrement symétrique*)

Le chiffrement à clé privée exige que toutes les parties qui sont autorisées à lire l'information aient la même clé que celle qui est utilisée pour le chiffrement des données.

Clef de chiffrement = clé de déchiffrement

Comme exemple d'algorithme à clé privée, on peut citer : *Kerberos, Data Encryption Standard, International Data Encryption Algorithmes...*

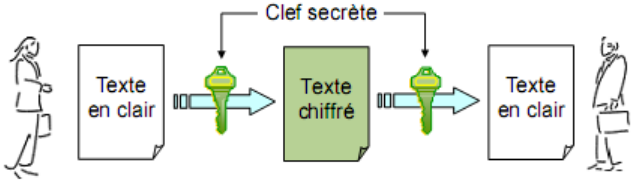


Figure 1.2 : Schéma représentant le chiffrement symétrique [06]

L'exemple ci-dessus, en considérant que les deux personnes sont A et B, on peut représenter cet échange par : A --> B : $\{message\}_{Kab}$

b. <u>Les algorithmes à clef publique :</u> (Chiffrement asymétrique)

Clef de chiffrement ≠ clé de déchiffrement

Le chiffrement asymétrique se base sur deux clés (*une privée et une autre publique*) pour chiffrer et déchiffrer les messages. Ces clefs sont distinctes et générées en même temps et elles dépondent étroitement l'une de l'autre, c'est-à-dire lorsqu'on chiffre avec l'une des clés, on doit forcément déchiffrer avec l'autre. Ainsi en utilisant la clef publique, tout le monde peut chiffrer un

message que seul le propriétaire de la clef privée pourra déchiffrer, et inversement, si on utilise la clef privée pour le chiffrement, tout le monde (*ceux qui possèdent la clé publique*) peut déchiffrer.

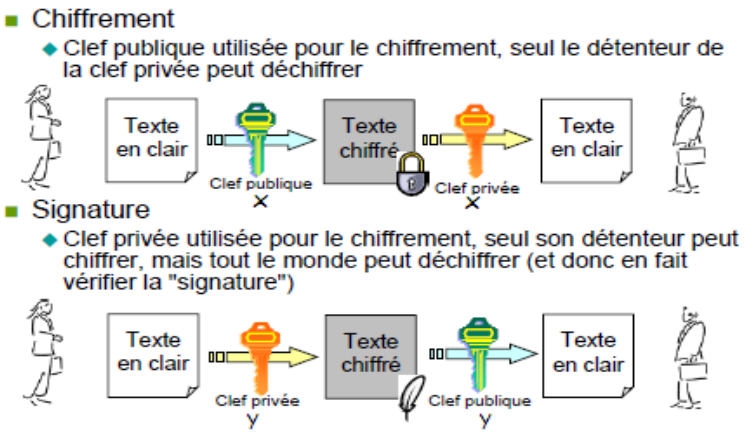


Figure 1.3 : Cryptographie à clef publique [06]

On peut identifier la provenance des données chiffrées par à la clef privée puisque une seule personne la possède, et donc lorsqu'une personne déchiffre le message avec sa clé publique, elle sait très bien d'où le message provient.

1.2. Intégrité et authenticité :

Authenticité = Authentification + Intégrité

Souvent on utilise le terme authentification afin de désigner l'authenticité, mais notez bien que l'authentification et l'intégrité sont inséparables. Lorsqu'un échange d'informations se présente au travers d'un canal de communication peu sûr, le destinataire aimerait bien s'assurer que le message s'émane de l'auteur auquel il est attribué et qu'il n'a pas été altéré pendant son voyage à travers le canal.

- **Authentification :** Consiste à s'assurer que les données s'émanent bien de l'expéditeur et non pas d'un autre utilisateur ou autre personne qui se prend pour l'expéditeur même.
- **Intégrité :** Consiste à s'assurer que les données n'ont pas été modifiées durant leur transfert.

Pour répondre à ces deux critères, les signatures et les certificats numériques sont apparus.

1.2.1. Signature numérique [1]

Un des avantages majeurs de la cryptographie à clé publique est qu'elle procure une méthode permettant d'utiliser des signatures numériques. Les signatures numériques permettent à la personne qui reçoit une information de contrôler l'authenticité de son origine, et également de vérifier que l'information en question est intacte. Ainsi, les signatures numériques des systèmes à clé publique permettent l'authentification et le contrôle d'intégrité des données. Une signature numérique procure également la non-répudiation, ce qui signifie qu'elle empêche l'expéditeur de contester ultérieurement qu'il ait bien émis cette information. Ces éléments sont au moins aussi importants que le chiffrement des données, sinon davantage.

Une signature numérique a le même objet qu'une signature manuelle. Toutefois, une signature manuelle est facile à contrefaire. Une signature numérique est supérieure à une signature manuelle sur le plan de l'authenticité d'une information. Ainsi, elle est pratiquement impossible à contrefaire, ce qui permet d'attester le contenu de l'information autant que l'identité du signataire.

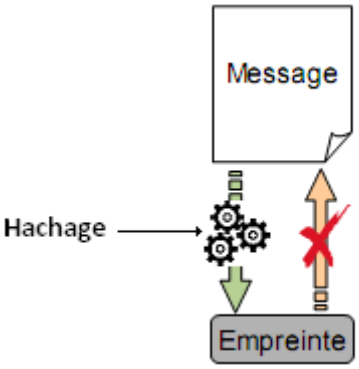
Sur le plan conceptuel, il est recommandé d'utiliser une clef privée car seul son possesseur pourra générer la signature et toute personne possédant la clef publique correspondante est apte à la vérifier (voir la figure 1.4).

1.2.2. Fonction de hachage

Lors d'échanges de messages chiffrés, il est important de pouvoir s'assurer que le message n'a pas été altéré ou modifié par un tiers pendant l'envoi. Les fonctions de hachage permettent alors de s'assurer de l'intégrité du message.

Aussi appelée fonction de condensation, une fonction de hachage est une fonction qui convertit une chaîne de longueur quelconque en une chaîne de taille inférieure et généralement fixe, la chaîne résultante est appelée ***empreinte*** (*digest* en anglais) ou condensé de la chaîne initial.

La fonction de hachage est une fonction à sens unique, c'est-à-dire qu'elle doit permettre de trouver facilement l'empreinte à partir du message, et d'empêcher de retrouver le message à partir de l'empreinte. Elle doit aussi être très sensible, pour qu'une petite modification du message entraîne une grande modification de l'empreinte. Autre



[1] La norme ISO 7498-2 définit la signature numérique comme étant des données rajoutées à une unité de données ou une transformation cryptographique d'une unité de données permettant à un destinataire de prouver la source et l'intégrité de l'unité de données en question (seul l'expéditeur est apte à générer la signature).

caractéristique d'une fonction de hachage, est qu'elle doit prendre en charge les collisions et savoir les gérer, c'est-à-dire qu'elle doit empêcher que deux messages différents aient la même empreinte.

En envoyant le message accompagné de son empreinte, le destinataire peut ainsi s'assurer de l'intégrité du message en recalculant le résumé à l'arrivée et en le comparant à celui reçu. Si les deux résumés sont différents, cela signifie que le message n'est plus le même que l'original, et qu'il a été altéré ou modifié.

Exemple : **MD5 (Message Digest 5)** qui fournie une empreinte de 128 bits.

SHA (Secure Hash Algorithm) qui donne une empreinte de 160 bits.

NB : *On utilise souvent le terme fonction de hachage pour désigner fonction de hachage à sens unique sans collisions.*

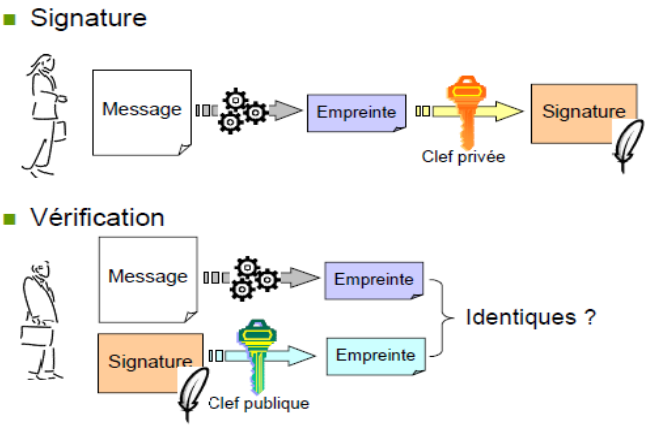


Figure 1.4 : Exemple de méthode de signature numérique [06]

1.2.3. Certificat électronique

Dans un environnement de clé publique, il est essentiel de s'assurer que la clé publique avec laquelle les données sont chiffrées, est celle du destinataire concerné et non une contrefaçon.

Un certificat numérique ou électronique fonctionne en gros comme une pièce d'identité matérielle.

Un certificat numérique [01] est une information attachée à une clé publique, et qui permet de vérifier que cette clé est authentique, ou valide. Les certificats numériques sont utilisés pour contrecarrer les tentatives de substituer une clé falsifiée à la clé véritable.

Un certificat numérique comporte trois éléments:

- Une clé publique.
- Une information de certification ("l'identité" de l'utilisateur, comme son nom, son adresse e-mail, etc.)
- Une ou plusieurs signatures numériques.

L'objet de la signature numérique sur un certificat est de garantir que les informations de certification ont été contrôlées par une autre personne ou organisme. La signature numérique ne garantit pas l'authenticité du certificat complet, elle garantit seulement que les informations d'identité ainsi signées correspondent bien à la clé publique à laquelle elles sont attachées.

- **Exemple de protocole cryptographique**

i. Le protocole Needham – Schroeder Public Key (NSPK)

On considère le protocole Needham-Schroeder (qui est un protocole d'authentification mutuelle). Deux entités sont impliquées A (initiator) et B (responder) dans ce protocole. La séquence des échanges de messages est la suivante :

1. A initie le protocole et envoie $\{Na.A\}_{K_b}$
2. B reçoit $\{Na.A\}_{K_b}$ et envoie $\{Na.Nb\}_{K_a}$
3. A reçoit $\{Na.Nb\}_{K_a}$ et envoie $\{Nb\}_{K_b}$
4. B reçoit $\{Nb\}_{K_b}$

$$A \rightarrow B \quad : \quad \{Na.A\}_{K_b}$$

$$A \leftarrow B \quad : \quad \{Na.Nb\}_{K_a}$$

$$A \rightarrow B \quad : \quad \{Nb\}_{K_b}$$

K_a et Kb sont les clés publiques de A et B
N_a et N_b sont des nombres aléatoires

Figure 1.5 : Représentation du protocole NSPK

ii. Intrus Dolev-Yao

On utilise le modèle Dolev-Yao pour modéliser le comportement d'un intrus.

L'intrus [16] a la capacité de capturer les messages ainsi que leurs injections dans le réseau.

Il est capable de générer de nouveaux messages en se basant seulement sur les messages qu'il possède. L'intrus peut composer :

- M.M' (concaténation de deux messages M et M') s'il connait M et M'.
- {M}_K s'il peut construire M et K.
- H(M) s'il connait la fonction de hachage H et le message M.
- N où N est un nonce[2]
- Etc.

iii. Attaque Man-In-The-Middle sur NSPK

L'attaque man-in-the-middle sur le protocole NSPK signifie qu'une personne malveillante *I* peut s'interposer entre deux entités (A et B) qui communiquent grâce à ce protocole. I(A) pour spécifier l'usurpation de l'identité de A par l'intrus.

1. A --> I : {Na.A}_Ki
2. I(A) --> B : {Na.A}_Kb

3. B --> A : {Nb.Na}_Ka
4. A --> I : {Nb}_Ki

[2] Les nonces sont traités comme des symboles dans le sens où ils ne sont pas décomposables ou calculables à partir d'autres atomes.

1. L'agent A commence par générer un nonce Na (qu'on appelle un nonce) qu'il concatène avec son identité et crypte le tout avec la clé publique de I avant de le lui envoyer.
2. A la réception du message par I, il le décrypte puis le crypte avec la clé publique de B et puis il le lui envoie.
3. A la réception du message, B déduit que A veut communiquer avec lui, en lui proposant un identifiant Na, et donc B génère lui aussi un nonce et renvoie le tout pour A.
4. A reconnait le nonce qu'il a envoyé pour I, il en déduit que le message vient de lui, et il lui confirme la bonne réception du message en lui renvoyant Nb.
   Et donc à la fin I connait le secret partagé entre A et B.

2. Le test logiciel

Le test logiciel est une activité du développement logiciel qui visent à accroitre la confiance dans le logiciel, cette confiance peut être recherchée [08] par :

- L'évaluation de la fiabilité du logiciel.
- L'établissement de la conformité avec une spécification.
- La détection des fautes dans le programme.

Le test logiciel est très couteux en termes de temps et d'argent, mais il est nécessaire, et en générale, plus une faute est détectée tard, plus elle coute chère. Selon [07], Le vol inaugural de la fusée Ariane 5, qui a eu lieu le 4 juin 1996 s'est soldé par un échec 40 secondes après le décollage, à la suite d'une panne du système de navigation. L'incident, dû à un bogue dans les appareils informatiques de navigation, a provoqué la destruction de la fusée ainsi que la charge utile : 4 sondes de la mission Cluster, d'une valeur totale de 370 millions de dollars, ce qui en fait le bogue informatique le plus coûteux de l'histoire.

Moralité : " il vaut mieux prévenir que guérir ! "

Il existe déférent types de test logiciel, qu'on peut illustrer par la figure 1.6 mais qui n'est pas exhaustive :

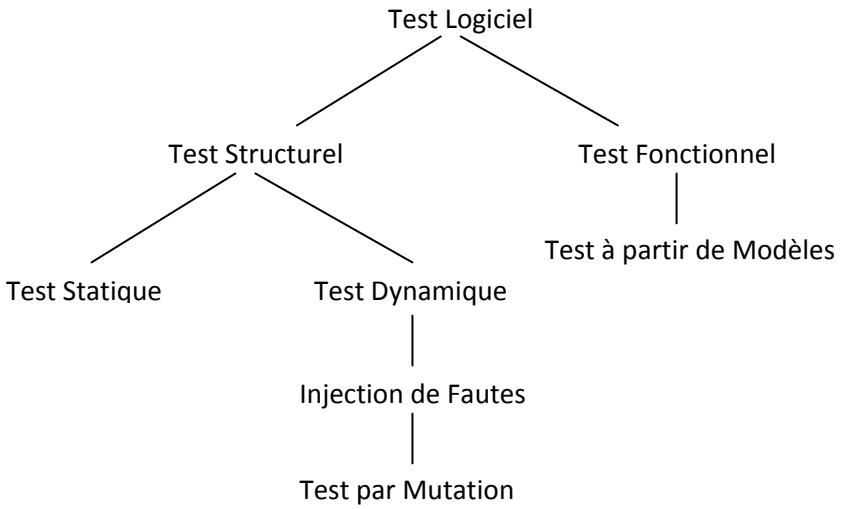


Figure 1.6 : Types de test logiciel

- **Test Structurel :** il se base sur l'analyse de la structure de l'application, appelé aussi test boite blanche.
- **Test Fonctionnel :** test de conformité par rapport à une spécification, connu aussi sous le nom de test boite noire.
- **Test Statique :** traite le texte du logiciel sans l'exécuter sur des données réelles.
- **Test dynamique :** dans ce genre de test : tester un programme signifie donner en entrée des valeurs et recevoir après exécution de celui-ci la ou les sorties pour les comparer avec les sorties attendues [09].
- **Test à partir de Modèles :** approche fonctionnelle de génération de tests basée sur la description du système à l'aide d'une spécification formelle.
- **Injection de Fautes :** consiste à modifier des instructions dans un programme, ce qui donne ce qu'on appelle mutants. Ce procédé n'est pas un type de test, mais une technique exploitée par le test par mutation.
- **Test par Mutation :** l'idée est de considérer des variantes du programme d'origine, appelées mutants, et de sélectionner un jeu de test afin de les détecter, ce principe sera l'objet de ce chapitre.

Dans ce qui suit, nous n'allons nous intéresser qu'au test par mutation, qui est le procédé de test utilisé dans ce projet/stage.

2.1. Test par mutation

Le test par mutation est une technique apparue grâce aux travaux de DeMillo et al. dans les années 1970 [26], elle se base sur l'injection de fautes dans des programmes qui donne naissance à des mutants et puis d'essayer de les trouver grâce un ensemble de données de test. Cette technique selon [27] couvre toutes les phases de développement ; des tests sur les spécifications aux tests unitaires en passant par les tests d'intégrations.

2.1.1. Techniques d'injection de fautes

L'injection de fautes est une technique qui permet de construire des parades logicielles aux défaillances identifiées. Elle est divisée en plusieurs étapes selon [28] :

- Identifier les zones à risques.
- Analyser les tests déjà effectués.
- Puis décider sur la nature des fautes à injecter et l'outil d'injection à utiliser.

L'injection de fautes peut se faire de différentes manières et à différents endroits, parmi eux, en peut citer :

- **Compile-time fault injection :** injection de fautes dans le programme à tester.
- **Runtime fault injection :** consiste à injecter des fautes dans l'environnement d'exécution du programme.
- **Fuzz testing :** tend à fournir au programme à tester des données invalides (inattendues). Puis on analyse le comportement du programme durant son exécution.

2.1.2. Génération des mutants

La génération des mutants consistent à prendre un programme, lui injecter une faute (une modification syntaxique). Cette dernière est généralement guidée par ce qu'on appelle opérateurs de mutations. Un opérateur mutationnel est une règle de génération de mutant à partir de programmes originels. Elle est désignée pour modifier les variables et les expressions des programmes, en remplaçant, ajoutant ou supprimant des opérateurs [27]. Une telle règle peut être par exemple le remplacement de l'opérateur addition (**+**) dans le programme **p** par l'opérateur soustraction (-), ce qui nous donne un autre programme syntaxiquement différent de l'originel, appelé mutant **p'**, comme illustré dans la figure 1.7 :

Programme **p** :	Mutant **p'** :
… If a = 0 then res := b **+** c ; return res; …	… If a = 0 then res := b - c ; return res; …

Figure 1.7 : Exemple illustrant la création d'un mutant.

2.1.3. Hypothèses du test par mutation

Pour que le test par mutation soit efficace, dans l'identification du jeu de test qui permet la découverte des fautes, un certains nombre de mutants doivent être créés afin de représenter les différentes fautes possibles pour le programme en question, or le nombre de fautes potentielles dans un programme est énorme [27] donc difficilement représentables. Le test par mutation prend juste un sous-ensemble des fautes possibles, celles qui font du programme erroné semblable à l'originel avec pour espoir que ce sous-ensemble sera suffisant pour simuler l'ensemble des fautes.

Cette théorie est basée sur deux hypothèses [26], la première appelée Hypothèse du programmeur compétent : elle suppose que le programmeur ne fait que des erreurs simples qui peuvent être corrigées par de simples modifications syntaxiques. La deuxième hypothèse s'appelle l'effet de couplage. À l'inverse de la première qui concerne le comportement du programmeur, cette hypothèse se base sur le type de fautes utilisé dans l'analyse mutationnelle. Elle dit que si une donnée de test permet de détecter un programme qui contient une faute simple, elle pourra aussi détecter des erreurs plus complexes (c'est-à-dire, détecter le même programme, mais cette fois avec plusieurs erreurs). Un programme qui contient une seule faute simple est appelé « mutant d'ordre 1 », celui qui contient deux est un mutant d'ordre 2 et celui qui contient n fautes est un mutant d'ordre n et ainsi de suite. Offut dans [29], [30] a confirmé cette supposition « l'effet de couplage » et a prouvé qu'un jeu de test fait initialement pour les mutants du premier ordre, tuait 99% de mutants de 2^{nd} et $3^{ème}$ ordre.

2.1.4. Processus de l'analyse mutationnelle

L'analyse mutationnelle comme illustrée à la figure 1.8, consiste à définir à partir d'un programme, un ensemble de mutants d'ordres 1. Un jeu de test est exécuté contre ces mutants, dans le but de les distinguer du programme originel. Un mutant qui donne des résultats différents de ceux du programme initial est immédiatement tué, par contre il existe des mutants qui donne les mêmes résultats que le programme initial [31], ils sont dit des mutants équivalents, qui ne sont pas détectables par le jeu de test. L'efficacité d'un jeu de test est définit par son score mutationnel *MS* qui est le rapport entre le nombre de mutants tués *DM* sur le nombre de mutants total *M* exemptés du nombre de mutant équivalents *E*.

$$MS = \frac{\mathrm{DM}}{\mathrm{M} - \mathrm{E}}$$

L'analyse mutationnelle a pour but d'améliorer le score mutationnel et le rendre le plus proche de 1, ce qui veut dire que le jeu de test est suffisant pour détecter l'ensemble des fautes dénotées par les mutants, ainsi une donnée de test a donc pour rôle de tuer les mutants, et plus elle en tue plus elle est fiable.

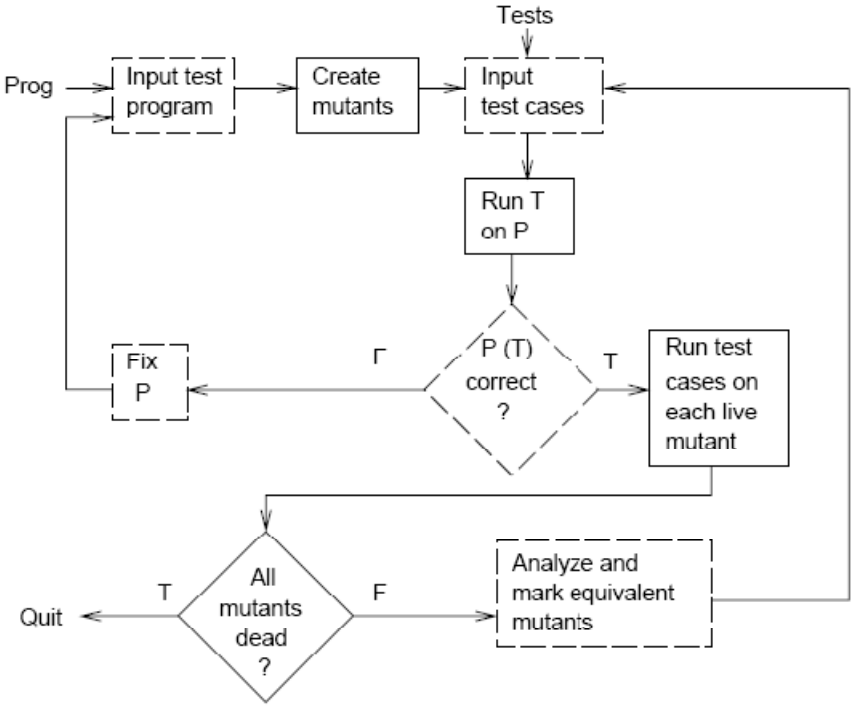


Figure 1.8 : Processus d'analyse mutationnel. [44]

Pour résumer, tester un programme revient à lui soumettre un ensemble de jeux de test dans le but d'atteindre un certain critère de test, et dans notre cas on l'appelle « mutation adequacy score » [27] ou qualité du test mutationnel qui est utilisée pour mesurer l'efficacité du jeu de test à détecter les fautes.

La partie la plus délicate dans le test est bien la génération des jeux de test. Après avoir déterminé à l'aide de critères de couverture l'ensemble des buts à atteindre, il faut rechercher des jeux de test qui permettent d'obtenir cette couverture [32].

Cette sélection peut être libre si on a un oracle complet (qui décide de la valeur de sortie pour toutes valeurs d'entrée) du programme sous test ou dans le cas contraire être contrainte par le manque d'information de l'oracle.

La technique de test par mutation souffre de certains problèmes tels que la difficulté d'appliquer le jeu de test sur le nombre important de mutants pour chaque programme, le problème des mutants équivalents qu'on ne peut pas détecter ainsi que le problème de l'oracle du test où il faut le faire à la main pour chaque donnée de test.

De nouvelle notion sont apparues pour parer à ces problèmes, parmi elles, on trouve : *strong mutation*, *weak mutation*, et *firm mutation* qui mélange les deux premières.

- ***Strong mutation ou mutation forte :*** cette notion [26] nous est un peu plus familière vue qu'elle fait référence à la mutation traditionnelle définit par DeMillo, c'est-à-dire que le mutant n'est tué que lorsqu'il donne des résultats différents de ceux du programmes initial à la fin de son exécution.
- ***Weak mutaion ou mutation faible :*** cette technique est apparue pour parer à la lourdeur de la mutation forte. Elle suppose que le programme initial est composé par un certain nombre de composants, et qu'un mutant est généré par la modification d'un seul composant. Le mutant est tué dès que le composant modifié a fini de s'exécuter, ce qui signifie qu'il n'est pas nécessaire d'attendre la fin de l'exécution du programme pour prendre la décision de tuer ou non le programme.
- ***Firm mutation :*** Woodward et Halewood [21] ont introduit une nouvelle technique de test comme étant un compromis entre la mutation faible et la mutation forte. Dans cette technique, un programme p est muté dans un certain point n et ainsi avoir un mutant p'. Le programme original p et le mutant p' sont comparés suivant un jeu de test J jusqu'à un nœud i. La mutation faible ainsi que la forte sont des cas particulier de la ''firm mutation''. Dans la faible, i = n et dans la forte, i est le nœud de sortie du programme i.e. le programme p' est exécuté jusqu'à la fin avant de rendre le résultat.

2.1.5. Le domaine d'applicabilité

2.1.5.1. Mutation des programmes

La mutation de programmes est appliquée sur le niveau unitaire, mais aussi sur le niveau intégration [33]. Les mutants sont générés pour représenter les erreurs aux seins des différents modules du programme, mais aussi pour représenter les fautes causées par la connexion ou l'interaction des différents modules du programme dans la phase d'intégration.

2.1.5.2. Mutation des spécifications

Le test par mutation était initialement proposé comme un « test boite blanche » qui couvre le niveau implémentation du cycle de développement du logiciel, mais il a fini par s'étendre sur l'ensemble du cycle de développement. La « spécification mutation » désigne la technique qui tente à rendre les spécifications plus solides et plus fiables. Elle a été introduite par Gopal et Budd en 1983 [34] [35]. Cette technique consiste à injecter des fautes dans les états du système ou dans les expressions logiques afin de générer des mutants de spécification. Ce dernier est désigné pour être tué s'il fournit des résultats non conformes aux objectifs attendus par le programme. Donc « spécification mutation » est utilisée [27] pour tester les fonctions qu'un logiciel est supposé remplir.

i. Le test par mutation pour les spécifications formelles

Les spécifications formelles peuvent être représentées de différentes façons, comme les machines à états finis, réseaux de pétri, diagrammes états-transitions… Les recherches antérieures concernant la mutation des spécifications étaient concentrées sur les spécifications de simples expressions logiques.

Plus récemment, plusieurs techniques formelles ont été proposées pour spécifier l'aspect dynamique des logiciels. Fabbri et al. [20] ont appliqué la mutation des spécifications pour valider des spécifications présentées comme des machines à états finis. Ils ont proposé 9 opérateurs mutationnels représentant les fautes liées aux états, événements et sorties d'un système à états finis. Les diagrammes états-transitions ou statecharts ont été largement utilisés pour les spécifications formelles de systèmes réactifs complexes. Les statecharts peuvent être considérés comme des extensions des machines à états finis, et le premier ensemble d'opérateurs mutationnels a été proposé toujours par Fabbri et al. en se basant sur leurs travaux antérieurs, et Yoon et al. ont utilisé ces résultats pour introduire un nouveau critère de test appelé ''State based Mutation Test Criterion'' (SMTC) qui est détaillé dans [19].

ii. Le test par mutation d'environnement d'exécution

Pendant le processus d'implémentation des spécifications, des bugs peuvent être introduit par les programmeurs à cause du manque de connaissances de l'environnement d'exécution souhaité. Ces bugs sont par fois difficiles à détecter et sont causés par les erreurs systèmes [19], la limitation en taille mémoire…

iii. Le test par mutation dans les réseaux

La robustesse des protocoles est un aspect important dans tous les réseaux.

Guoqiang Shu et David Lee dans leurs travaux [25] et afin de tester les protocoles cryptographiques de point de vue confidentialité des messages, ont modélisé les protocoles de sécurité comme des machines à états/transitions et les failles de sécurité comme une fonction de mutation qui a comme modèle de faute ''l'absence de garde'' dans les transitions de ces systèmes comme illustré sur la figure 1.9.

Exemple : Soit la spécification d'un protocole représenté par l'automate (a). Et une fonction de mutation $\delta_{PA}(S_1, Y, t,[a=1])$.

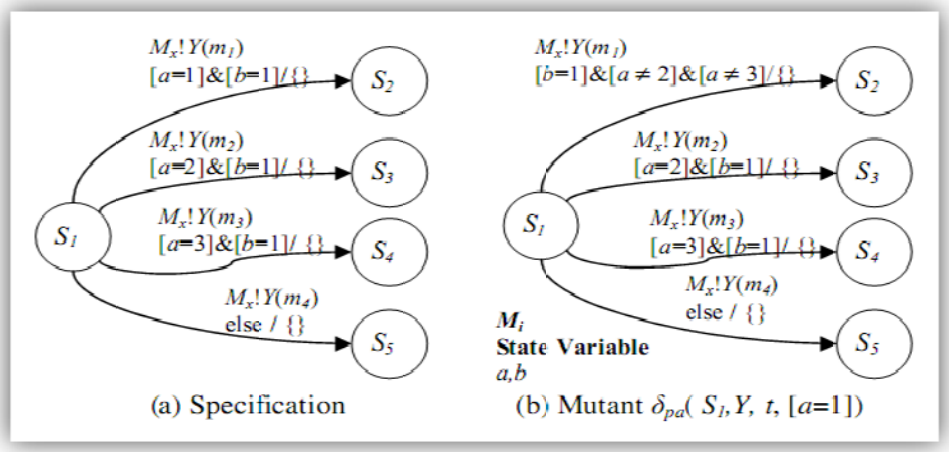


Figure 1.9 : Application d'une fonction de mutation sur une spécification

Une suite de test est générée de telle sorte que pour chaque mutant, il existe au moins une séquence de test qui le détecte en comparant les résultats avec ceux de la spécification correcte.

3. Présentation de l'outil AVISPA

3.1. Définition

AVISPA (Automated Validation of Internet Security Protocols and Applications) [11] est un outil d'analyse et de validation automatisées des applications et des protocoles de sécurité.

AVISPA fournit un langage formel modulaire et hautement expressif de modélisation des protocoles de sécurité et de spécification des propriétés recherchées. Ces spécifications sont écrites à l'aide du langage de spécification de protocole de haut niveau appelé **HLPSL** (High Level Protocol Spécification Language) inspiré de TLA (Temporal Logic of Actions).

AVISPA se distingue des outils existants par les nouvelles fonctionnalités qu'il intègre. Il propose une analyse entièrement automatique, et recouvre un large spectre de protocoles ; il est capable de combiner plusieurs techniques de vérification. Ces

différentes techniques [10] reposent soit sur la vérification de modèles, soit sur la résolution de contraintes extraites par exploration « symbolique » des protocoles.

L'insécurité d'un protocole se détecte en vérification de modèles par la possibilité d'atteindre un état où, par exemple, le mot de passe apparaît en clair. Avec la méthode de résolution de contraintes il s'agit de tester si l'Intrus peut décrypter un message avec les informations qu'il a en sa possession, ou qu'il peut l'acquérir en se faisant passer pour un autre agent.

3.2. Architectures de l'outil AVISPA

Cet outil prend en entrée une spécification écrite sous format HLPSL [13], cette dernière est traduite en un format de plus-bas niveau appelé format intermédiaire (**I**ntermediate **F**ormat **IF**) à l'aide d'un traducteur appelé **hlpsl2if Translator**[3]**.** Le fichier résultant est ainsi directement interprété et la vérification de la spécification du protocole par les back-ends peut commencer.

AVISPA dispose de quatre différents back-ends comme illustré sur la figure 1 et qui implémentent des techniques d'analyse allant de la falsification de protocole (découverte d'attaque sur le protocole d'entrée) à des méthodes de vérification à base d'abstractions pour des nombres fini ou infini de sessions. Si une propriété de sécurité est violée dans la spécification, l'analyseur produit la trace de séquence d'événement qui mène à une faille et affiche quelle propriété est violée.

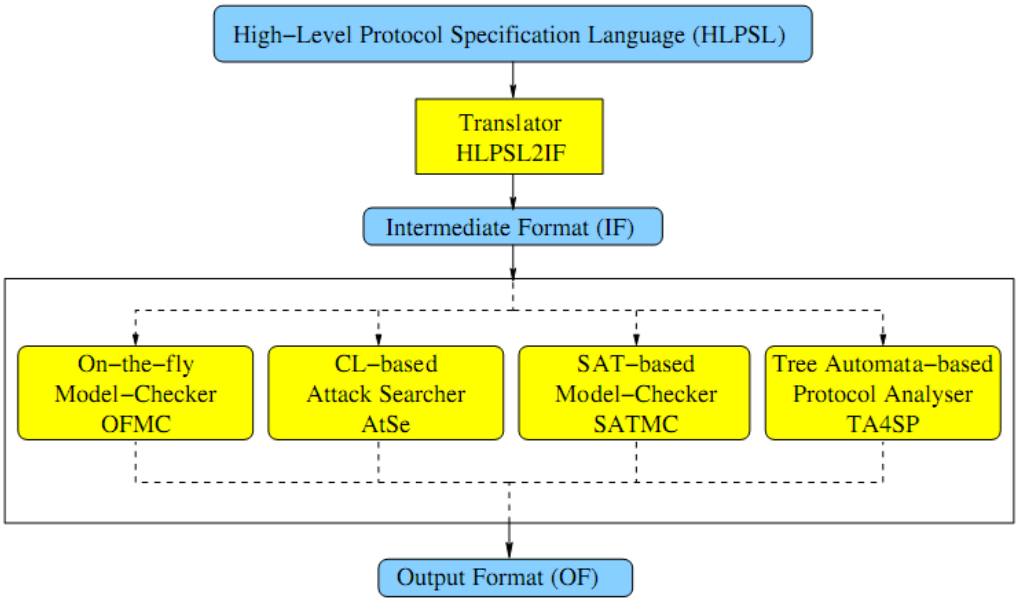


Figure 1.10 : Architecture de l'outil AVISPA [12]

[3] À noter que cette transformation est transparente pour l'utilisateur car le traducteur est automatiquement invoqué**.**

3.2.1. On-the-fly Model-Checker (OFMC)

Historiquement [16], l'histoire d'OFMC a commencé au sein du projet AVISS, et puis OFMC a mûri au sein du projet AVISPA.

Il effectue une vérification bornée en explorant le système de transition décrit par une spécification IF [14]. OFMC implémente des techniques symboliques correctes et également complètes. Il supporte la spécification des opérateurs à propriétés algébriques tels que le OU exclusif ou encore l'Exponentielle.

3.2.2. Constraint logic- based Attack Searcher (CL-AtSe)

C'est un outil basé sur des techniques de résolution de contraintes et implémente une procédure de décision décrite dans [17]. Il permet de faire une traduction d'une spécification d'un protocole de sécurité sous forme de relations de transition au format IF, vers un ensemble de contraintes qui peuvent être utilisées pour trouver des attaques sur le protocole en question [12]. Les deux méthodes (la traduction et la vérification sont totalement automatiques et prises en charge par l'outil CL-AtSe sans intervention d'outils externes.

Les possibilités de CL–AtSe ont été étendues lors du projet AVISPA pour supporter des opérateurs possédant des propriétés algébriques (**Xor** ou **Exp**).

3.2.3. SAT-based Model-Checker (SATMC)

Cet outil a été développé au laboratoire DIST à Gènes (Italie) [16]. Il construit une formule propositionnelle codant un déploiement borné du système de transition de IF [12][15], l'état initial et l'ensemble des états représentant la violation des propriétés de sûreté spécifiées en IF (a fortiori en HLPSL). La formule propositionnelle est ensuite donnée à résoudre à un solveur SAT, sélectionné parmi ces quatre : zCHAFF, mCHAFF, SIM et SATO. Ensuite, tout modèle satisfaisant cette formule est retourné sous forme d'attaque.

3.2.4. Tree-Automata-based Protocol Analyzer (TA4SP)

La particularité de cet outil de vérification est qu'à partir d'un état initial il fait soit une sur-approximation ou une sous-approximation des connaissances de l'intrus en utilisant des automates d'arbres [12]. Cette méthode permet de savoir si un certain état est accessible ou non et que l'intrus peut savoir certaines connaissance ou non et ainsi de conclure l'absence d'attaque sur le secret pour des scénarios exécutés un nombre indéterminé de fois.

3.3. Utilisation de l'outil AVSPA suivant [12]

- On commence par la spécification du protocole à tester grâce au langage HLPSL, ainsi que les propriétés à vérifier.
- On lance AVISPA à l'aide d'une invite de commandes toute en précisant l'analyseur (back-end) qu'on va utiliser et par défaut AVISPA utilise OFMC back-end.
- Ensuite AVISPA, après analyse, il déclare que soit le protocole est sûr (mais peut être sous certaines conditions), ou bien le protocole présente des failles et dans ce cas, on pourra changer la spécification du protocole et poursuivre depuis l'étape 2.

3.4. Présentation du langage HLPSL

Les protocoles qui sont destinés à être tester par l'outil AVISPA, sont spécifiés en HLPSL qui est inspiré de TLA (Temporal Logic of Actions). Il permet ainsi la représentation d'un protocole de sécurité par un système d'états/transitions sur lequel la vérification des propriétés de sûreté exprimées en logique temporelle linéaire sera effectuée.

Une spécification HLPSL décrit des systèmes de transition qui représentent les différentes entités intervenant dans ce protocole. Le fichier résultant est un fichier dont l'extension est «. hlpsl » et qui sera traduit par la suite en fichier dont l'extension « .if » à partir de laquelle les différents back-ends effectuent leurs vérifications.

Le langage HLPSL est basé sur la définition des rôles :

- **rôles basiques :** qui vont représenter le comportement de chaque participant intervenant dans ce protocole.
- **rôles de composition :** qui représentent les scénarios des rôles basiques i.e. cette catégorie permet l'instanciation des différents rôles basiques pour modéliser la totalité du protocole.

3.5. Structure d'une spécification HLPSL

Une spécification HLPSL est composée de trois parties : Une liste de définition de rôles, une liste des objectifs ou des propriétés de sûreté à vérifier, et enfin, l'instanciation du rôle principal (généralement sans arguments).

3.5.1. Définition des rôles

Les rôles peuvent être définis comme des processus indépendants qui ont des noms, reçoivent des informations en paramètre et contiennent des déclarations locales. On distingue deux catégories de rôles :

3.5.1.1. Les rôles basiques

Cette catégorie décrit la connaissance initiale et le comportement de chaque entité (agent) intervenant dans le protocole spécifié. Ci-dessous, un exemple simplifié d'une déclaration de rôle pour l'agent ''alice''. Cet exemple est donné afin de mieux comprendre cette notion de rôle.

La connaissance initiale liée à chaque rôle (dans notre exemple, concerne alice) est exprimée par une liste de paramètres qui sont : la clé symétrique, les différents agents connus par alice, un ensemble de canaux qui seront utilisés pour la communication. Dans la déclaration d'un rôle, on peut trouver une clause optionnelle **''played_by''** pour spécifier quel agent joue le rôle considéré.

Exemple :

```
% ici, on déclare les commentaires
role alice (A, B: agent, Kab : semmetric_key,
                    Snd, Rcv : channel(dy))
played_by A def =
local State : nat,
      Na, Nb: text,
      Kab: semmetric_key
init State := 0
transition
State = 0 /\ Rcv(Kab.{Nb}_Kab) =|> State' := 1
...
end role
```

Nous pouvons trouver aussi une section qui contient une liste de variables locales et leurs types, qui peuvent être initialisées dans la section **init**. Une liste de transitions est définie comme dans la toute dernière partie du rôle. Cette dernière va définir le comportement de l'agent à travers de des transitions qu'il peut faire suivant une certaine condition pour changer d'état.

Sur notre exemple, si l'agent A se trouve à l'état (State=0) et que sur le canal **Rcv** il peut lire le message ''Kab.{Nb}_Kab'' alors la transition est déclenchée et change d'état vers State' = 1. (State' signifie que l'ancienne valeur stockée dans State sera écrasée par la nouvelle valeur qui est "1").

3.5.1.2. Les rôles de composition

Cette catégorie sert à définir les scénarios des rôles basiques, et permet de combiner les autres rôles soit en parallèle ou en séquence, et définir ainsi le protocole en tant que session.

Chaque spécification doit avoir un rôle spécial qui peut s'appeler par exemple session et qui va représenter une session du protocole. Dans l'exemple ci-dessous est représentée une session du protocole NSPK entre deux agents alice et bob, et leurs rôles sont joués respectivement par A et B.

Exemple :

```
role session (A, B: agent,
                    Ka, Kb: public_key)
def =
composition
          alice(A, B, Ka, Kb) /\
          bob(A, B, Ka, Kb)
end role
```

Il existe un autre rôle de composition [16] couramment appelé **environnement** ou **rôle principal.** Ce rôle ne possède aucun paramètre et sert à définir l'état initial du système en précisant la connaissance initiale de l'intrus par la clause **intruder_knowledge,** ainsi que qu'un nombre fini de rôles session.

Exemple :

Cet exemple décrit une session qui se déroule entre deux agents a et b qui ont une clé commune kab. Et initialement l'intrus connait les deux agents a et b.

```
role environment()
def=
  const  a,b : agent,
              kab : symmetric_key
intruder_knowledge = {a,b}
              composition
session(a,b,kab)
end role
```

3.5.2. Déclaration des propriétés de sûreté

Les propriétés de sûreté du protocole à évaluer par AVSPA telles que le secret, l'authentification sont déclarées dans une section à part et afin de les vérifier, il est nécessaire d'introduire dans la spécification des éléments événementiels appelés **signaux**. En HLPSL, il existe différents signaux permettant d'exprimer les propriétés de sûreté [12] [13] :

- **secret(E, id, S) :** déclare que l'information E est un secret partagé par un ensemble S d'agent.
- **witness(A, B, id, E) :** pour la propriété d'authentification (faible) de l'agent A auprès de B grâce à la donnée E. Cet objectif sera identifié par la constante id dans la section réservée à la déclaration des propriétés de sécurité.
- **Request (B, A, id, E) :** pour l'authentification forte entre A et B. Elle déclare que B demande une vérification de la valeur E. La fonction de id est la même que pour witness.
- **wrequest(B, A, id, E) :** elle est similaire à **request()**, mais cette fois pour l'authentification faible.

La déclaration des objectifs ou des propriétés à vérifier se fait dans une section à part. Chaque propriété est identifiée par une constante qui réfère le prédicat défini (secret, witness, request et wrequest) pour une transition donnée.

Exemple :

```
State = 0 /\ RCV(start) =|> State' := 1 /\ M' := new()
                          /\  SND(Kab.{M'}_Kab)
                          /\  secret (M', id_sec, {A, B})
```

L'exemple ci-dessus permet de déclarer la nouvelle valeur de M après exécution de la transition comme étant un secret partagé entre A et B. L'instruction M' := new() permet de générer aléatoirement une nouvelle valeur pour M.

3.5.3. Instanciation d'un rôle

La création d'une instance d'un rôle est comme l'appel d'une procédure, en donnant une valeur pour chaque paramètre déclaré dans le rôle, et bien sûr le nombre d'arguments ainsi que leurs types doit être les mêmes que ceux des paramètres formels du rôle. Pour le rôle principal, il suffit d'invoquer son nom (usuellement sans paramètre comme suit : **environment()**).

3.6. Exemple de spécification en HLPSL

Soit le protocole suivant

1. A --- > B : na
2. B --- > A : {na}_K

La clé K est partagé entre les deux agents A et B. Après la deuxième transition, la clé K est mise à jour dans le rôle B puis dans A. avec cette formule K' := H(K).

La spécification de ce protocole en HLPSL est la suivante :

```
role server ( A,B: agent, K: symmetric_key,
              H: hash_func, Snd,Rec: channel(dy))
played_by A
def=
    local  State       : nat,
           Na          : text
    init State := 0
    transition
    1.   State = 0 /\ Rec(start) =|> State' := 1
                       /\ Na' := new() /\ Snd(Na')
    2.   State = 1 /\ Rec({Na}_K) =|> State' := 2
                       /\ K' := H(K)/\
             request(A,B,auth_client,{Na}_K)
  end role
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
role client ( B,A: agent,K: symmetric_key,
              H: hash_func, Snd, Rec: channel(dy))
played_by B
def=
    local  State          : nat
           Na             : text
    init State := 0
    transition
    1.   State = 0 /\ Rec(Na') =|> State' := 1
                  /\ Snd({Na'}_K) /\ K' := H(K) /\
                  witness(B,A,auth_client,{Na'}_K)
end role
```

```
role session(A,B : agent,
             K : symmetric_key,
             H: hash_func)
def=
  local St,Rt,Sl,Rl : channel(dy)
  composition
             client(B,A,K,H,St,Rt)
             /\ server(A,B,K,H,Sl,Rl)
end role
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
role environment()
def=
  const a,b : agent,
        k : symmetric_key,
        h : hash_func,
  auth_client:protocol_id
  intruder_knowledge = {a,b,h}
  composition
        session(a,b,k,h)
end role
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
goal
    authentication_on auth_client
end goal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
environment()
```

4. Principe de l'analyse lexico-syntaxique

Le domaine de résolution d'un problème donné est généralement très différents de celui dans lequel un ordinateur travaille, d'où la nécessité d'un langage comme ''intermédiaire'',

et dans notre cas, nous allons nous intéresser au langage HLPSL (pour la spécification des protocoles cryptographiques).

Un langage informatique (dans notre cas le langage HLPSL) est un formalisme de représentation de la connaissance (les protocoles cryptographiques). La forme de ses phrases est décrite par une grammaire[4], la syntaxe du langage, son intérêt est de véhiculer une sémantique, les aspects lexicaux-syntaxiques ne sont qu'un moyen pour y parvenir.

Afin de mener à bien notre projet, nous allons écrire un analyseur lexico-syntaxique en java (parser en anglais) pour le langage HLPSL, et pour cela, nous allons utiliser le générateur de parser **JavaCC** (**Java C**ompiler **C**ompiler). Mais avant, nous allons expliquer la méthode classique pour l'écriture de parser [22].

On distingue deux fonctions pour les parsers :

4.1. L'analyse lexicale

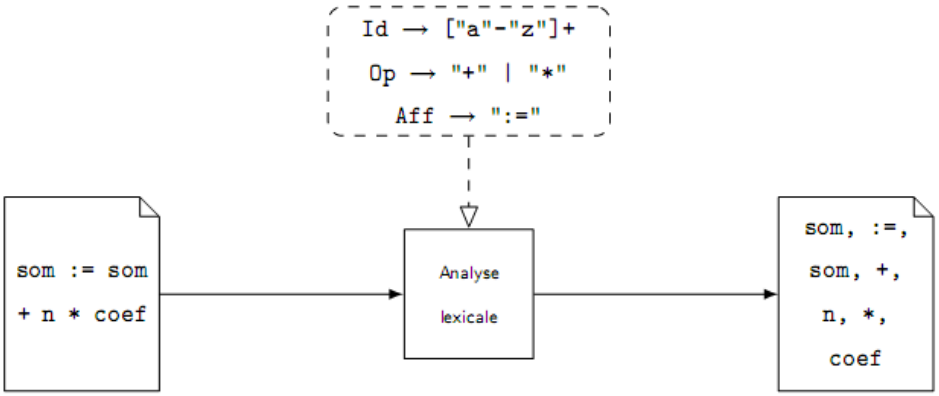


Figure 1.11 : Analyse lexicale [23]

Son rôle consiste à lire le programme source, caractère par caractère, et à produire une en sortie, une séquence d'unités lexicales (les symboles terminaux) dites « tokens » et qui sera traitée par l'analyseur syntaxique.

NB : Cette analyse ignore les blancs et les commentaires.

- **Exemple :** L'analyse lexicale du fragment de programme suivant :

  begin i := 10 ; end

  regroupe les caractères en unités suivantes :
  - Le mot clé ''begin''
  - L'identificateur ''i''
  - Le symbole d'affectation '':=''
  - La constante entière ''10''
  - Le séparateur '';''
  - Le mot clé ''end''

[4] La grammaire du langage HLPSL est donnée en annexe.

Un analyseur lexical [22] peut être vu comme une fonction qui à chaque fois qu'elle est appelée par l'analyseur syntaxique, lit le programme source jusqu'à reconnaitre le symbole suivant qu'elle retourne alors à l'appelant. La lecture commence avec le caractère suivant le dernier token reconnu et se termine avec la reconnaissance du plus long préfixe du reste du programme source constituant un symbole. Les symboles sont décrits par des expressions régulières, ils sont alors reconnus par des automates d'état fini.

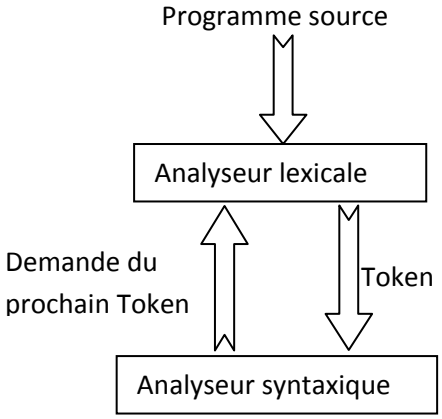


Figure 1.12: Analyse lexico-syntaxique[08]

Token, modèle et attribut : certaines chaines de caractères d'un programme source ont une même structure et une même fonction syntaxique. On les regroupe alors en une classe dite token (unité lexicale).

Cette classe est décrite par une expression régulière dite ''*modèle du token*''. On dit que le modèle filtre chaque chaine de la classe.

Un lexème : est une séquence de caractères filtrée par le modèle d'un token, autrement, c'est un élément d'un token (attribut).

- **Conception d'un analyseur lexical [22]**

Une conception méthodique et efficace passe par certaines étapes que nous allons les donner sans rentrer dans les détails.

i. Déterminer les différents tokens du langage, ainsi que leurs modèles.

ii. Pour chaque modèle, construire un automate d'état fini indéterministe (**N**on deterministic **F**inite **A**utomaton, **NFA**).

iii. Construire le NFA global qui reconnaît tous les tokens.

iv. Construire l'automate d'état fini déterministe (DFA) global équivalent au NFA global et le minimiser.

v. Associer les actions à entreprendre au niveau des états finaux, celles-ci peuvent être :

- Reculer d'une position sur le programme source.
- Calculer la valeur d'un attribut.
- Retourner un couple <token, attribut>
- etc

vi. et enfin implémenter le DFA global minimal en incluant les actions attachées aux états finaux.

4.2. L'analyse syntaxique

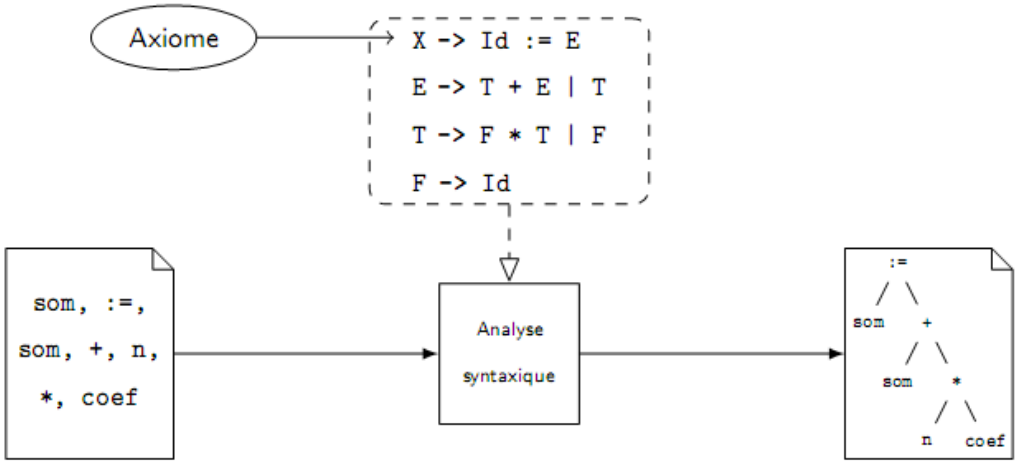


Figure 1.13 : Analyse syntaxique [23]

Le rôle de cette phase consiste à vérifier si la séquence de symboles (tokens) retrouvée par la phase précédente est conforme à la syntaxe du langage décrit par une grammaire algébrique. Une séquence de tokens fait partie du langage si seulement si elle peut être obtenue par dérivation à partir de l'axiome, c.-à-d. s'il existe un arbre de dérivation pour cette séquence.

Le problème de l'analyse syntaxique revient donc toujours à chercher un arbre de dérivation.

Les principales méthodes d'analyse syntaxique connues sont résumées dans le schéma :

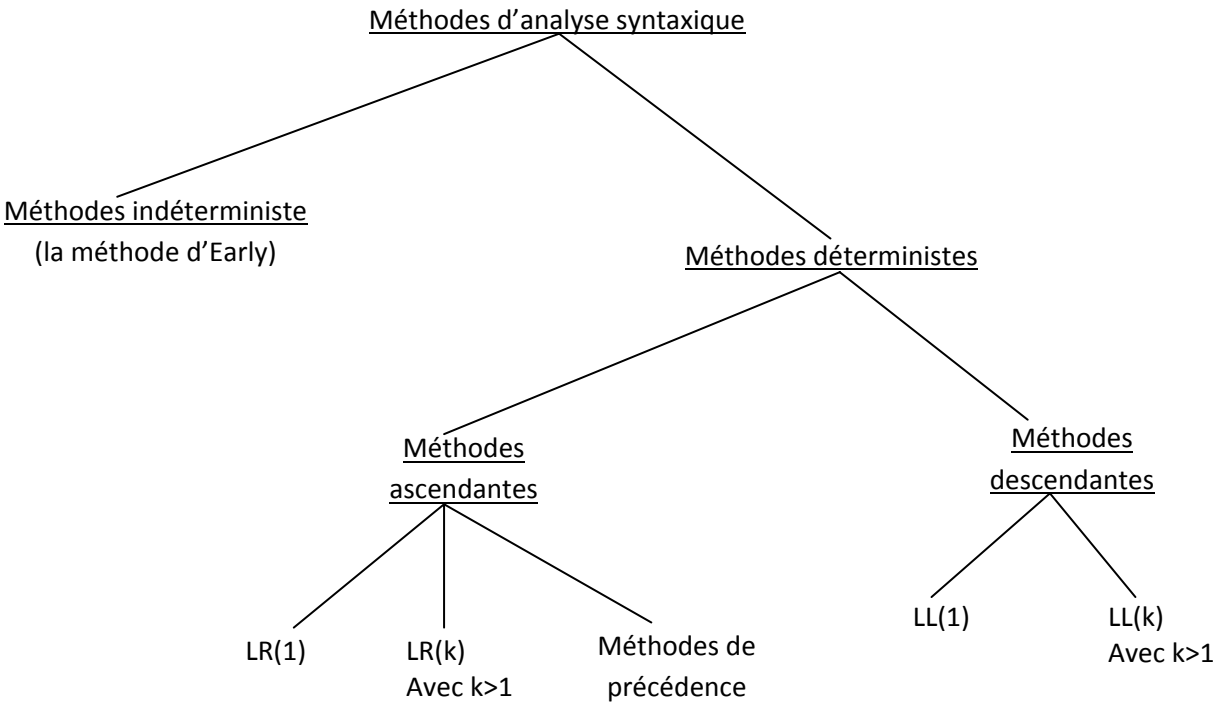


Figure 1.14 : Méthodes d'analyse syntaxique [08]

4.3. Rappels sur les grammaires algébriques

Dans une grammaire algébrique (grammaire de type 2 ou hors contexte), toutes les règles de production sont de la forme [24][22]:

$A \rightarrow \alpha$ avec $\alpha \in (V_N \cup V_T)^*$ V_N : ensemble des symboles non-terminaux
V_T : ensemble des symboles terminaux

- **Dérivation :** est une suite de dérivation élémentaire qui n'est rien d'autre que l'emploi d'une règle de production de gauche à droite (on remplace le non-terminal de gauche par le membre droit).
  - **o** Dérivation gauche (la plus à gauche « left most ») est une dérivation dans laquelle le non-terminal le plus à gauche est toujours dérivé en premier.

  Exemple : soit la production suivante

  $E \rightarrow E + E \mid E \times E \mid num$

  La dérivation la plus à gauche pour le mot num + num x num se ferait comme suit :

  $E \Rightarrow E + E \Rightarrow num + E \Rightarrow num + E \times E \Rightarrow num + num \times E \Rightarrow num + num \times num$

- **Grammaire ambigüe**

  Une grammaire G est ambigüe [22] s'il existe au moins un mot appartenant au langage engendré par G, (L(G)) et admettant deux ou plusieurs arbres de dérivation.
  Sur l'exemple précédent, la grammaire ($E \rightarrow E + E \mid E \times E \mid num$) est ambigüe car il existe un mot « 2 + 5 x 7 » qui a deux arbres de dérivation.

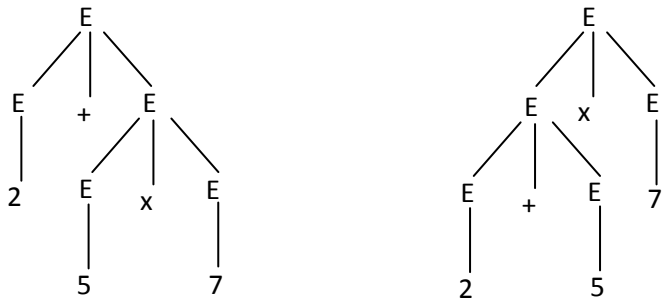


Figure 1.15 : Arbres de dérivation du mot 2+5x7

- **Récursivité à gauche**

  Une grammaire contient une récursivité à gauche directe (immédiate) si elle contient au moins une règle de production de la forme : $A \rightarrow A\alpha$

  Pour éliminer ces récursivités à gauche directes, on remplace toute règle de la forme : $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$ (les β_i ne commencent pas par A)

  par : $A \rightarrow \beta_1 B \mid \beta_2 B \mid \ldots \mid \beta_m B$
  $B \rightarrow \alpha_1 B \mid \alpha_2 B \mid \ldots \mid \alpha_n B \mid \varepsilon$

- **Analyse syntaxique descendante**

  Le principe des méthodes d'analyse descendantes [22] est de tenter de trouver la dérivation la plus à gauche de la séquence de tokens, cela revient à tenter de construire l'arbre de dérivation en commençant de la racine et en construisant les nœuds dans un pré-ordre.

  Les méthodes d'analyse syntaxique descendantes les plus répondues sont : la descente récursive et la méthode prédictive.

4.4. L'analyseur lexico-syntaxique JavaCC

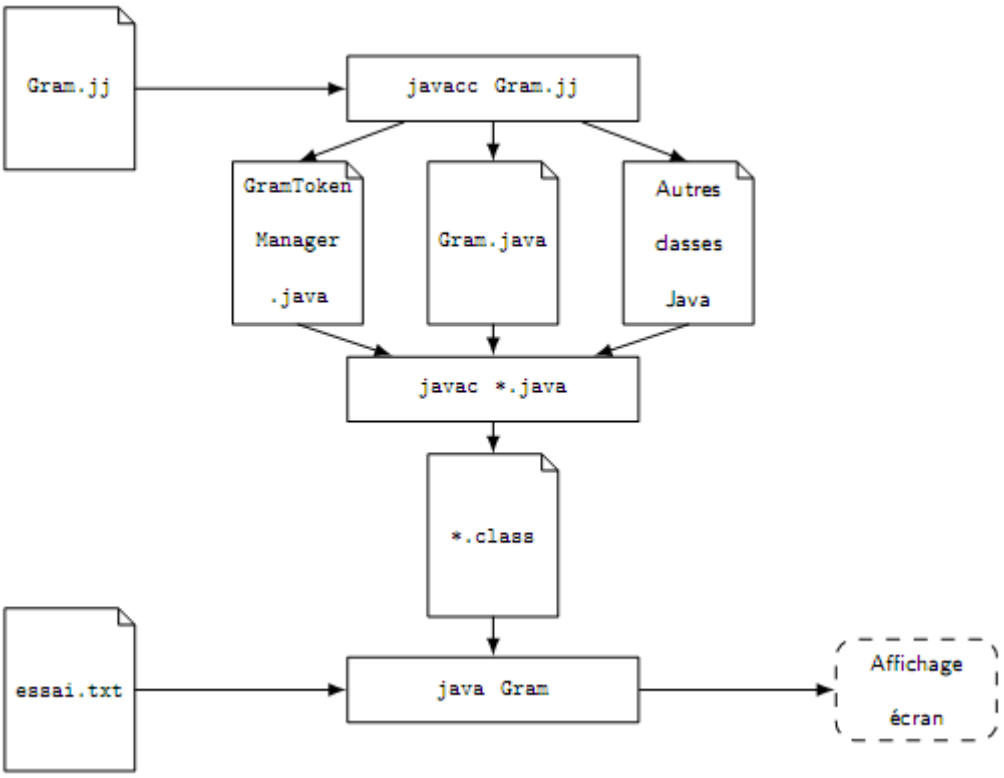


Figure 1.16 : Mise en œuvre de JavaCC [23]

JavaCC signifie **Java C**ompiler **C**ompiler [36], son rôle est de générer des analyseurs lexico-syntaxiques en Java à partir d'une grammaire écrite dans une forme bien définie et enregistrée sous l'extension « .jj ».

Copié dans Gram.java

```
PARSER_BEGIN(Gram)
public class Gram {
    public static void main (String arg[]) throws ParseException {
        Gram p = new Gram(System.in);
        p.axiome();
    }
}
PARSER_END(Gram)
```

Liste de règles lexicales et syntaxiques

```
TOKEN : {
    <SALUT : "Bonjour" | "Hello">
}

void axiome() : {
}{
<SALUT> unNonTerminal() ("\n"|"\r")* <EOF>
}
void unNonTerminal() : {
}{ ...
}
```

Figure 1.17 : Structure du fichier gram.jj [23]

Nous allons utiliser aussi le préprocesseur JJTree, qui prend une grammaire sous forme « .jjt » semblable à « .jj » (syntaxe JavaCC + annotations JJTree(#)[5]) et génère un fichier « .jj » et un certain nombre de classes de nœuds.

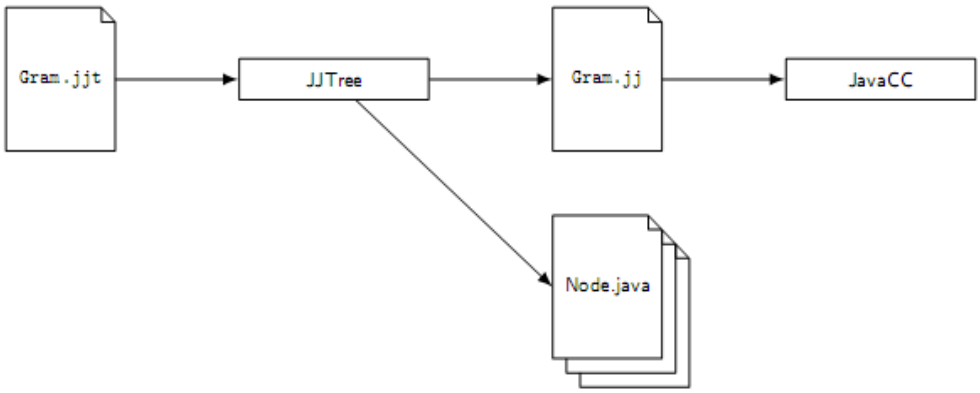


Figure 1.18 : Préprocesseur JJTree [23]

[5] Annotations syntaxiques de créations de nœuds

III. Contributions

Introduction

Dans le cadre de ce projet, et pour le mener à bien, nous avons proposé certaines modifications qui seront appliquées à nos spécifications en ''.hlpsl''. Ces modifications dans les programmes que nous appellerons fonctions de mutation seront développées en détails dans ce qui suit.

Pour un souci de clarté, nous proposons de partitionner notre travail en plusieurs points que nous allons développer un par un.

Les exemples de protocoles qui illustreront par la suite nos propositions, sont essentiellement basés sur les travaux de Véronique Cortier et al. dans [02].

Avant d'entamer les explications et les propositions de mutation, nous donnons un schéma qui résume la démarche qu'on a adoptée et qui met en évidences les points clés où nous allons appliquer nos modifications et enfin on va les implémenter à l'aide du langage JAVA.

1. Propositions de mutation

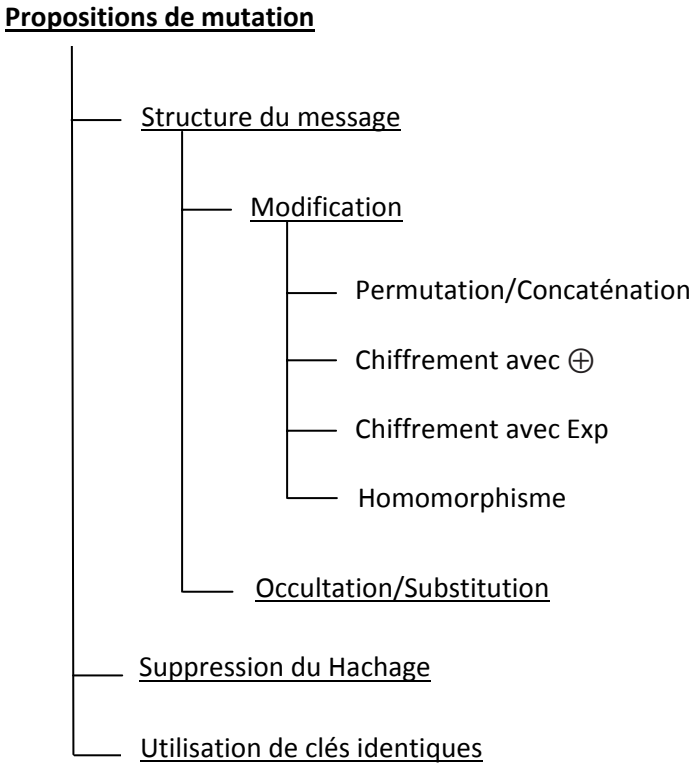


Figure 2.1 : Résumé des propositions de mutation

1.1. Actions sur la structure des messages

1.1.1. Modification

Les modifications peuvent être appliquées à la structure du message reçu ou émis de plusieurs manières. Parmi elles, nous distinguons

1.1.1.1. Permutation / Concaténation

Nous proposons ici une mutation qui concerne une propriété algébrique qui est la commutativité.

En effet, certains protocoles prennent en charge cette propriété c'est-à-dire autorise la permutation et concaténation des éléments constituants le message chiffré, et ça peut engendrer des failles de sécurités.

NB : Une fonction F est dite commutative si : F(x,y)=F(y,x).

A titre d'exemple, nous donnons la spécification du protocole ***NEEDHAM-SCHRODER-LOWE MODIFIED PROTOCOL***. Electivement, le premier message envoyé dans la version originale est {Na,A}_PK(B). Le fait de permuter le nonce Na avec l'identité A, engendre une attaque [18] que nous allons décrire par la suite. Mais avant, voici la spécification de ce protocole.

Spécification du protocole

```
A, B : principal
Na, Nb : fresh number
PK, SK : principal --> key (keypair)
     1.   A -> B : {A,Na}_PK(B)
     2.   B -> A : {[Na,Nb],B}_PK(A)
     3.   A -> B : {Nb}_PK(B)
```

Une attaque sur ce protocole serait celle où un intrus établit simultanément deux sessions. La première (''session i '') est établie avec B en usurpant un agent A, et la deuxième (''session ii '') avec l'intrus (avec sa véritable identité) et l'agent A.

Cette attaque est rendue possible grâce notamment à la propriété d'associativité de la fonction « the pairing function » qui permet d'identifier les éléments constituants le message (ici, le message est composé de deux éléments).

i.e.

```
[[I, Nb], B] = [I, [Nb, B]]      (associativité)
```

L'attaque

```
i.1.  I(A) -> B : {A,I}_PK(B)
i.2.  B -> I(A) : {[I,Nb],B}_PK(A)
ii.1.      I -> A : {I,[Nb,B]}_PK(A)
ii.2.      A -> I : {[[Nb,B],Na'],A}_PK(I)
i.3.  I(A) -> B : {Nb}_PK(B)
```

Dans ce travail, nous proposons de faire une permutation sur des éléments se trouvant dans le message émis ou bien reçu. Cela est valable pour les protocoles ''non-typés'', ainsi que les protocoles ''typés'' mais en faisant attention à ne permuter que les éléments de même type.

Par exemple, dans un programme représentant un protocole en hlpsl, nous pouvons remplacer : RCV(Na,Nb,B) par RCV(Nb,Na,B).

i.e :

{ canal(x,y,A) --> canal(y,x,A) , canal(A,x,y) --> canal(A,y,x)}

1.1.1.2. Chiffrement avec ⊕ « OU Exclusif »

Avec cette technique, nous cherchons à remplacer la méthode de cryptage par une autre qui utilise le OU Exclusif ou bien XOR qui a certaines propriétés algébriques.

$x \oplus (y \oplus z) = (x \oplus y) \oplus z$	(associativité)
$x \oplus y = y \oplus x$	(commutativité)
$x \oplus 0 = x$	(élément neutre)
$x \oplus x = 0$	(nilpotence)

Donc, nous allons chercher dans nos spécifications en hlpsl tout ce qui est de la forme $\{x\}\_Ka$ et le remplacer par $\{x\}\oplus Ka$.

Les protocoles utilisant le XOR, sont sensibles à quelques attaques, par exemple le protocole ***BULL'S AUTHENTICATION*** qui a la spécification suivante :

```
A, B, C, S : principal          (A, B et C sont des agents et S est un serveur)
Kab, Kbc, Na, Nb, Nc : fresh number
Kas, Kbs, Kcs : symkey          (clés symétriques)
h : message, symkey --> message

A calcule Xa = h([A,B,Na],Kas), [A,B,Na]
   1.   A -> B : Xa
B calcule Xb = h([B,C,Nb,Xa],Kbs), [B,C,Nb,Xa]
   2.   B -> C : Xb
C calcule Xc = h([C,S,Nc,Xb],Kcs), [C,S,Nc,Xb]
   3.   C -> S : Xc
   4.   S -> C: A, B, Kab ⊕ h(Na,Kas), {A,B,Na}_Kab,
           B, A, Kab ⊕ h(Nb,Kbs), {B,A,Nb}_Kab,
           B, C, Kbc ⊕ h(Nb,Kbs), {B,C,Nb}_Kbc,
           C, B, Kbc ⊕ h(Nc,Kcs), {C,B,Nc}_Kbc
   5.   C -> B: A, B, Kab ⊕ h(Na,Kas), {A,B,Na}_Kab,
           B, A, Kab ⊕ h(Nb,Kbs), {B,A,Nb}_Kab,
           B, C, Kbc ⊕ h(Nb,Kbs), {B,C,Nb}_Kbc
   6.   B -> A: A, B, Kab ⊕ h(Na,Kas), {A,B,Na}_Kab
```

Ce protocole décrit dans [03] tente d'établir une session avec des clés symétriques fraiches entre trois agents. A la fin de l'exécution de ce protocole, chacune des paire d'agents {A,B} ou {B,C} partage respectivement une clé symétrique Kab ou Kbc.

Une attaque qui utilise les propriétés du XOR serait par exemple : si l'agent C est malveillant, il garde les messages Kab ⊕ h(Nb,Kbs) et Kbc ⊕ h(Nb,Kbs) envoyés par S à l'étape 4. Et dès qu'il connait Kbc, il peut calculer « Kbc ⊕ Kab ⊕ h(Nb,Kbs)⊕ Kbc ⊕ h(Nb,Kbs)» et déduire Kab grâce aux propriétés de XOR.

Il existe aussi la même faille sur le protocole **NSPK-XOR**

```
A -> B: {A,Na}_Kb
B -> A: {Nb,XOR(B,Na)}_Ka
A -> B: {Nb}_Kb
```

Une attaque serait comme suite avec I comme agent malveillant :

```
A -> I : {A,Na}_Ki
I -> B : {A,XOR(I,XOR(B,Na))}_Kb
B -> A : {Nb,XOR(B,XOR(I,XOR(B,Na)))}_Ka
A -> I : {Nb}_Ki
```

Donc A lorsqu'il reçoit le message {Nb,XOR(B,XOR(I,XOR(B,Na)))}_Ka il calcule XOR(Na,XOR(B,XOR(I,XOR(B,Na)))) pour extraire l'identité de l'émetteur et trouve que c'est I. donc A lui envoie le nonce Nb qui n'est connu normalement que par A et B.

1.1.1.3. Chiffrement en utilisant « Exp »

Comme dans le cas précédant où nous avons considéré un chiffrement avec XOR, ici nous considérons le chiffrement avec la fonction exponentielle. Mais avant, nous donnerons quelques caractéristiques de cette fonction :

$exp(x+y) = exp(x) * exp(y)$

$exp(x) * exp(-x) = 1$

$exp(exp(x,y),z) = exp(x,y*z)$

$exp(exp(x,y),z) = exp(exp(x,z),y)$

Nous allons présenter le célèbre protocole ***DIFFIE HELLMAN KEY EXCHANGE*** qui utilise les propriétés de l'exponentiel pour la définition d'une clé symétrique.

La fonction Kap est définit comme suit :
kap(P, X, Y) = exp(X, Y) mod P Et **kap(P,kap(P,G,Y),X) = kap(P,kap(P,G,X),Y)**

Sa spécification est la suivante :

```
A, B : principal
P, G, Na, Nb : fresh number
kap : number, number, number --> number
  1.    A -> B : P, G
  2.    A -> B : kap(P,G,Na)
  3.    B -> A : kap(P,G,Nb)
```

Avec P : un nombre premier et G : primitive root modulo P [1]

Une attaque sur ce protocole serait qu'un intrus peut usurper l'identité d'un autre agent. Et dans l'exemple qui suit l'intrus I se fait passer pour l'agent B.

```
1.      I(A) -> B : P,G
2.      I(A) -> B : kap(P,G,Ni)
3.      B -> I(A) : kap(P,G,Nb)
```

1.1.1.4. Homomorphisme

Cette propriété est annoncée comme suite : {[x, y]}_k = [{x}_k, {y}_k].

Cette propriété est particulièrement satisfaite lors de l'utilisation du chiffrement par bloc comme dans « **Electronic Code Book ''ECB'' ».** Ce mécanisme ECB [04] représente la façon la plus évidente d'étendre le chiffrement par bloc sur un texte de longueur quelconque. Il divise le texte clair en blocs de taille fixe n (généralement n = 64) et chiffre chacun d'eux séparément, après avoir éventuellement complété le dernier bloc afin d'atteindre la taille n.

[1] Si G est une primitive root modulo P alors pout tout entier a avec gcd(a,P) = 1, il existe un entier k tel que : $G^k \equiv a \pmod P$

Afin d'exploiter cette propriété pour notre travail, nous proposons de remplacer tout les messages de la forme {x, y, z}_k par {x}_k, {x}_k, {x}_k. Et afin d'illustrer l'importance de cette propriété, nous donnerons un exemple de protocole qui l'utilise, ainsi d'une attaque sur ce même protocole à cause justement de cette propriété. Ce protocole est ***NEEDHAM-SCHROEDER-LOWE PROTOCOL WITH ECB*** et sa spécification est la suivante :

```
A, B : principal
Na, Nb : fresh number
PK, SK : principal -> key (key pair)
     1.   A -> B : {Na, A}_PK(B)
     2.   B -> A : {Na, Nb, B}_PK(A)
     3.   A -> B : {Nb}_PK(B)
```

Attaque.

Un intrus en jouant deux sessions (`i` et `ii`) du protocole, peut extraire `{Na, Nb}_PK(A)` de `{Na, Nb, B}_PK(A)`. Il le réutilise avec `{I}_PK(A)` pour calculer `{Na, Nb, I}_PK(A)` et force ainsi A à lui décrypter `Nb`.

```
i.1.       A -> I :{Na, A}_PK(I)
ii.1.      I(A) -> B :{Na, A}_PK(B)
ii.2.      B -> I(A) :{Na, Nb, B}_PK(A)
```

L'intrus `I` extrait `{Na, Nb}_PK(A)` et calcule `{Na, Nb, I}_PK(A)`.

```
i.2.       I ->A :{Na, Nb, I}_PK(A)
i.3.       A ->I :{Nb}_PK(I)
ii.3.      I(A) -> B :{Nb}_PK(B)
```

Notre travail ici consiste à remplacer tout les messages de la forme

1.1.2. Occultation/substitution

Cette partie va concerner la substitution d'un message ou d'une partie d'un message par une autre. A titre d'exemple : RCV({Na.Nb.A}) sera remplacé par RCV({Na.X}). Cette propriété est particulièrement intéressante, vu qu'il existe des protocoles qui s'intéressent seulement à une partie du message reçu, sans par exemple vérifier l'identité de l'émetteur chiffrée dans le message, et ça conduit facilement à des attaques par ''rejeu''.

1.2. Suppression de la fonction de hachage

Dans cette proposition, nous allons chercher tout simplement à supprimer toute fonction de hachage qui figure dans la spécification du protocole (c-à-d. manipuler les messages sans l'utilisation du hachage) i.e. [h(x) --> x].

1.3. Session avec des clés publiques identiques

Dans ce cas, nous allons exécuter une session d'un protocole avec les mêmes clés publiques identiques. Par exemple, nous utiliserons session(A,B,K,K) lors de l'exécution du protocole, c'est-à-dire les deux agents A et B ont la même clé publique K.

➢ **Récapitulation**

Le tableau ci-dessous résume toutes les propositions de mutation que nous avons présentées précédemment, ainsi que l'endroit où elles peuvent s'appliquer i.e. il nous informe sur la façon dont la mutation va sera appliquée, ie si la mutation va s'appliquer sur une paire de messages ou bien à des endroits particuliers, et dans ce cas là (pour un souci de clarté), une note de bas de page est créée pour chaque mutation ponctuelle.

Mutation ponctuelle	Paire envoi/réception
Chiffrement avec $\oplus$ [2]	Permutation/Concaténation
Chiffrement avec Exp [2]	Occultation/Substitution
Homomorphisme [2]	
Suppression du Hachage [2]	
Utilisation de clés identiques [3]	

2. Implémentation

Pour entamer le deuxième point de ce chapitre qui est la phase d'implémentation de notre modèle de faute, rappelons la démarche globale adoptée pour mener à bien ce projet.

- Définition des différents patterns ou des signatures représentant les endroits où nos mutations (*que nous avons définit précédemment*) seront appliquées. Cette étape sera traitée par la suite dans ce chapitre.
- Génération de l'arbre syntaxique AST qui représente la spécification du protocole cryptographique (à l'aide de l'outil ***jjtree*** et ***javacc*** [4]).
- Et puis, parcours de l'arbre par de différents programmes à la recherche des signatures pour l'application des mutations souhaitées. Rappelons que pour chaque nœud de l'arbre correspond une classe en java qui la représente[5]. Le programme qui parcours l'arbre est connu sous le nom d'un visiteur, il possède par convention un nom qui se rapporte au ***traitement effectué*** par le visiteur concaténé avec le mot "***Visitor***".

[2] Ce type de mutation s'applique dans tous les points où le chiffrement ou la fonction de hachage choisie apparait.

[3] Ce type de mutation sera appliqué en un seul point particulier.

[4] Reportez vous au chapitre précédent pour plus de détails sur cette étape.

[5] La classe qui représente un nœud porte le même nom que ce dernier précédé par "AST".

Pour mieux comprendre cette dernière étape, nous rappelons brièvement l'idée de base de la technique de programmation du patron visiteur (*En anglais* "***visitor design pattern***"*).*

2.1. Le patron visiteur

Les "Design patterns" visent à rendre les systèmes orientés objet plus flexibles. En particulier, le pattern visiteur [42] permet la définition d'une nouvelle opération sur un objet sans changer la structure des classes. Un visiteur selon [38] est le nom d'une des structures de patron de conception comportemental. Il représente une manière de séparer un algorithme (le visiteur) d'une structure de données (dans notre cas un AST, Arbre de Syntaxe Abstraite représentant une spécification d'un protocole cryptographique).

Un visiteur possède une méthode par type d'objet traité. Pour ajouter un nouveau traitement, il suffit de créer une nouvelle classe dérivée de la classe Visiteur. On n'a donc pas besoin de modifier la structure des objets traités, contrairement à ce qu'il aurait été obligatoire de faire si on avait implémenté les traitements comme des méthodes de ces objets.

En général [39], le pattern visiteur peut être utilisé dans toute application où il ya des structures de données qui doivent être interprétées de différentes façons. En effet, il peut être utilisé pour parcourir les structures de données sous forme d'arbre [41] et Ainsi, modifier le comportement de cette hiérarchie de classes sans avoir à changer ces dernières. Il suffit pour cela, que les objets de la structure se laissent traverser en acceptant le visiteur (qui contient les opérations à effectuer). La figure 2.2 nous montre à travers un exemple la manière d'utiliser cette technique de conception.

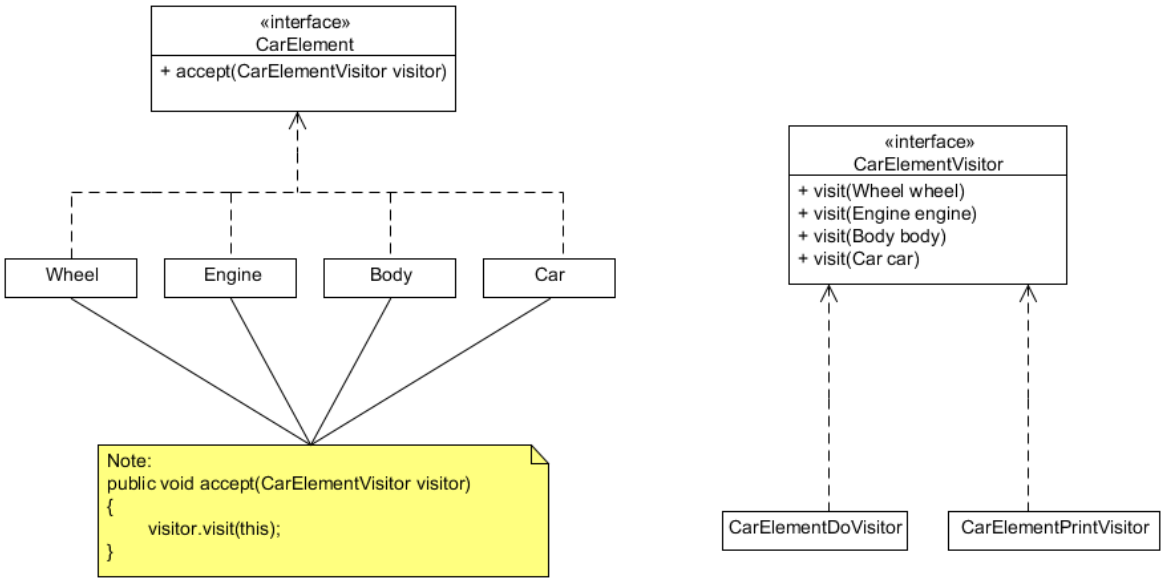


Figure 2.2 : exemple d'utilisation du "Visitor pattern" [37]

Pour résumer, et selon [41], la méthode du « pattern visitor » comprend les catégories suivantes:

- ***Visitor***

  C'est une interface pour les visiteurs. Il déclare des méthodes de visite nommé visit(nom_de_la_classe_à_visiter). Dans la figure précédente, il s'appelle « CarElementVisitor ».

- ***Concrete Visitor***

  Il représente une classe pour chaque opération sur les données. Il implémente l'interface Visitor, ainsi que chacune des méthodes de visite déclarée dans cette dernière. Sur notre figure, ça concerne les deux classes « CarElementDoVisitor » et « CarElementPrintVisitor ».

- ***Data/Element***

  Ceci est une interface qui décrit le type de données (par exemple l'interface CarElement sur la figure). Il déclare une méthode accept(Visitor) qui prend un objet visitor comme argument.

- ***Concrete Data / Concrete Element***

  Ceci représente toute les classes qui implémente l'interface « Data / Element » et dont leurs instances seront manipulées par les différents visiteurs. Chacune de ces classes implémente la méthode accept(Visitor) qui appelle la méthode visit(nom_de_l'element_à_visiter) adéquate de l'objet visitor.

Vous pouvez trouver plus d'informations dans [40] sur l'architecture et le fonctionnement du pattern visiteur.

L'avantage majeur de cette technique [6] est dû au fait que le pattern visiteur propose une solution sans réflexivité, pour déterminer quelle méthode invoquer en fonction du type dynamique de l'objet courant, comme illustré à la figure 2.3.

Comme les fils de la structure AST peuvent avoir des types différents et que l'interprète Java base sa recherche de méthodes sur les types statiques et non dynamiques des arguments [43], ces appels récursifs ne peuvent pas être directs.

En effet, la méthode visit appelée serait sinon celle du type statique (type connu à la compilation) du fils. Une indirection dans la classe de l'objet doit donc être effectuée pour que la méthode visit correspondant au type dynamique soit invoquée. Cette indirection est réalisée par l'appel de la méthode accept, de l'objet à visiter et qui délègue l'exécution à la méthode visit appropriée dans l'objet visiteur.

[6] Cette technique, comme expliquée précédemment, offre la possibilité d'ajouter de nouvelles opérations sans modification ou recompilation des classes des objets de la structure.

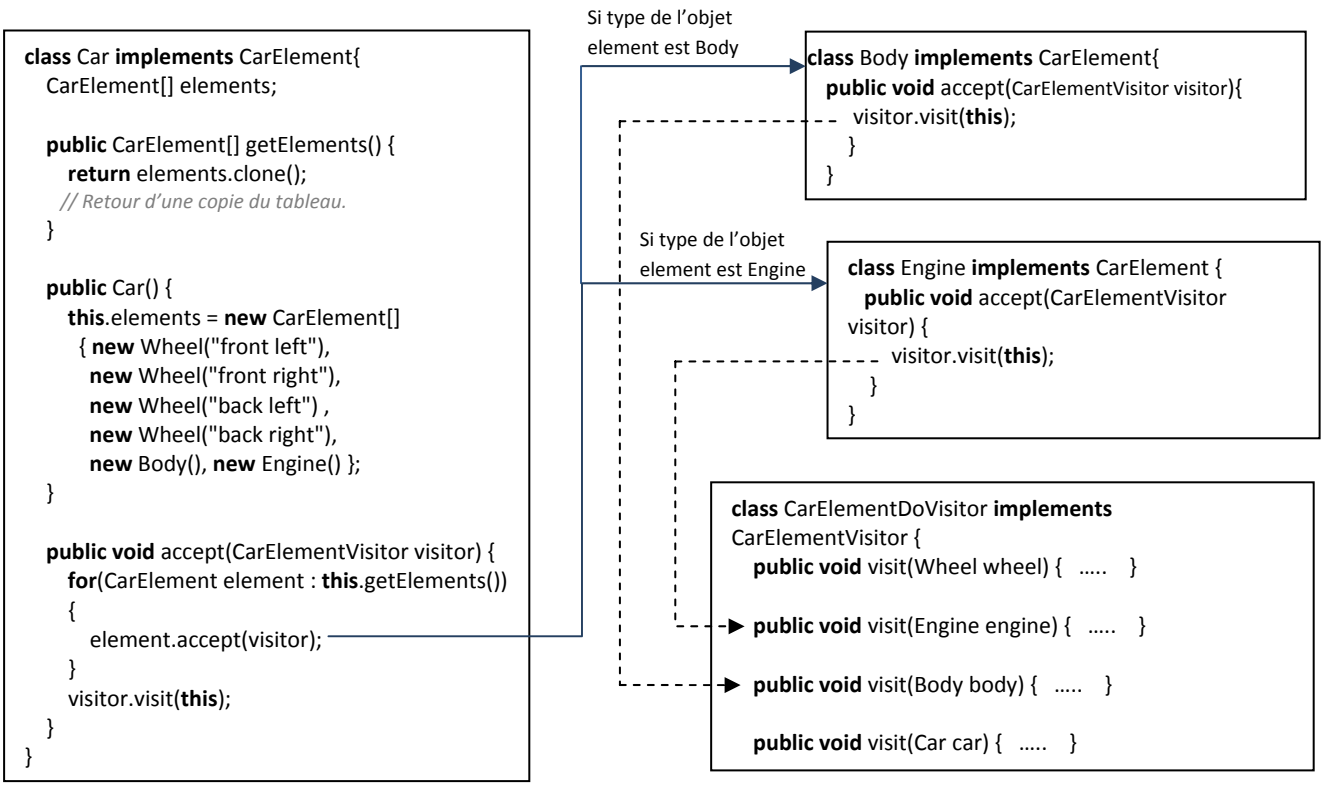


Figure 2.3 : Mécanisme d'indirection du patron de conception visiteur.

2.2. Définition des différents patterns pour les mutations

Nous avons présenté précédemment les différentes mutations à appliquer sur nos spécifications de protocoles, et pour cela, nous aurons besoin de bien repérer les emplacements, où les mutations sont susceptibles être appliquées. Nous donnerons ici, le ou les patterns (motifs) qui correspondent à chaque mutation.

Afin de bien suivre, nous proposons de découper la section qui suit (c'est-à-dire l'implémentation des propositions de mutation) en deux sous sections, suivant que la mutation est réalisée suivant "***une***" ou "***deux***" passes. Et nous donnerons par la même occasion des rappels sur ces mutations définies auparavant, et qui concerne donc, le cryptage (avec l'opérateur XOR et EXP), la concaténation/permutation (des messages envoyés/reçus), décomposition du message crypté en plusieurs autres cryptés avec la même fonction (mutation concernant l'homomorphisme), suppression des fonctions de hachage, occultation/Substitution (d'une partie du message par une autre), affaiblissement de la garde des transitions et enfin utilisation de clés publiques identiques (pour tous les agents qui participent à la même session de communication).

2.2.1. Les mutations avec une passe

Ce genre de mutation sera appliqué en une seule exécution du visiteur, qui va analyser la spécification, identifier les endroits possibles pour mutation et enfin, application de cette dernière.

2.2.1.1. Le cryptage par XOR et EXP

Pour ce type de mutation, le cryptage est réalisé à partir du nœud "Encryption" qui a comme fils directs, suivant la grammaire du langage, deux fils (virtuels) soit "Subtype_of" et "Bracketed_subtype_of" soit "Expression" et "Bracketed_expression", comme illustré sur la figure 2.4. Mais concrètement sur notre arbre AST, pour localiser l'emplacement des mutations, nous allons nous intéresser juste au nœud " Encryption " car ses nœuds fils qu'on a évoqués précédemment ne sont pas créés mais remplacés par leurs fils respectifs, ce qui nous fait une explosion combinatoire du nombre de motifs à repérer sur l'arbre. Ce type de mutation va s'appliquer sur la totalité de la spécification du protocole et pas seulement sur une partie.

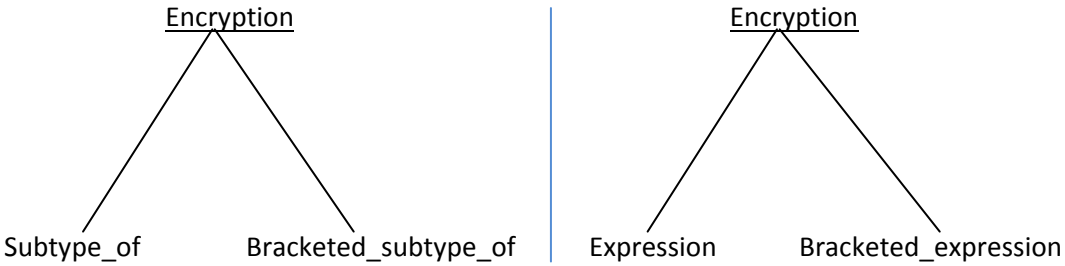


Figure 2.4 : les patterns correspondants au cryptage

A titre d'exemple, prenons un fragment d'une spécification du protocole nspk.hlpsl suivant :

```
2. State = 2 /\ RCV({Na.Nb'}_Ka) =|> State':=4 /\ SND({Nb'}_Kb) /\ request(A,B,alice_bob_nb,Nb')
```

Ce fragment représente une transition définie dans l'un des rôles du protocole, et nous allons juste nous intéresser à la partie « RCV({Na.Nb'}_Ka) ».

Après avoir passé ce protocole dans notre parser, son arbre AST sera généré, et le fragment correspondant au RCV({Na.Nb'}_Ka) est le suivant :

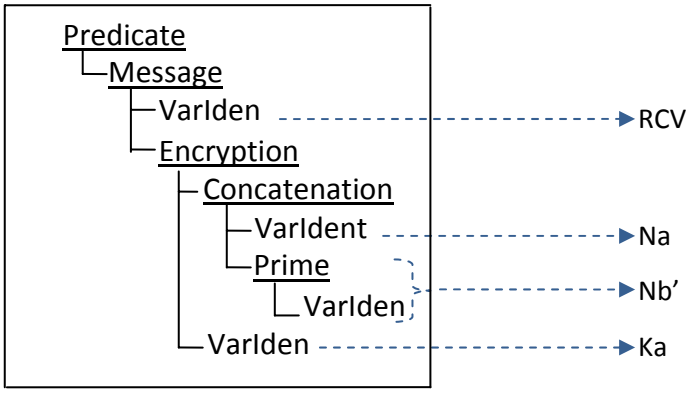


Figure 2.5 : Fragment de l'arbre AST correspondant à « RCV({Na.Nb'}_Ka) »

Le nœud « Encryption » est représenté par la classe ASTEncryption qui contient une méthode jjtAccept qui va déléguer l'exécution à la méthode visit appropriée dans l'objet visiteur. Cette classe est déclarée comme suit :

```
public class ASTEncryption extends SimpleNode {
  public ASTEncryption(int id) {    super(id);   }
  public ASTEncryption(parserHLPSL p, int id) {
    super(p, id);
  }
  /** Accept the visitor. **/
  public Object jjtAccept(parserHLPSLVisitor visitor, Object data) {
    return visitor.visit(this, data);
  }
}
```

Dans le cas où le visiteur ne fait aucune modification du programme source, la méthode visit qui concerne le nœud « Encryption » va reproduire le même message crypté. Cette classe est représentée ci-contre :

```
public Object visit(ASTEncryption node, Object data) {
        String s = " {";
        s += node.jjtGetChild(0).jjtAccept(this,data);
        s += "}_";
        s += node.jjtGetChild(1).jjtAccept(this,data);
        return s;
  }
```

Mais dans le cas d'un visiteur qui va effectuer une mutation avec l'opérateur XOR, une telle classe serait comme suit :

```
public Object visit(ASTEncryption node, Object data) {
        String s = " xor(";
        s += node.jjtGetChild(0).jjtAccept(this,data)+", ";
        s += node.jjtGetChild(1).jjtAccept(this,data)+") ";
        return s;
  }
```

De cette façon, à la fin de l'exécution de notre visiteur, au lieu qu'il nous retourne notre expression originale « RCV({Na.Nb'}_Ka) », il va nous imprimer «RCV(xor(Na.Nb',Ka)».

Remarque : toutes les méthodes de manipulation des nœuds sont regroupées et déclarée dans une seule interface appelée « Node ». Cette dernière est implémentée par la classe « SimpleNode » qui sera la superclasse de toutes les classes qui représentent l'arbre AST.

2.2.1.2. Homomorphisme

Dans ce cas, on cherchera à reproduire la propriété suivante

[{ msg1 **.** msg2 }_key] = [{msg1}_key **.** {msg2}_key]

Comme expliqué précédemment, pour ce type de mutation, nous n'allons nous intéresser qu'au nœud « Encryption » dont la classe correspondante accepte d'être traverser par le visiteur en délégant l'exécution à la méthode visit qui lui correspond. Cette méthode est la suivante :

```
public Object visit(ASTEncryption node, Object data) {
        String s = "";
        if (node.jjtGetChild(0).toString() == "Concatenation"
         && node.jjtGetParent().toString() != "Encryption"){
                for (int i=0; i < node.jjtGetChild(0).jjtGetNumChildren(); i++){
                        s += " {";
                        s += node.jjtGetChild(0).jjtGetChild(i).jjtAccept(this,data);
                        s += "}_"+node.jjtGetChild(1).jjtAccept(this,data);
                        if(i < node.jjtGetChild(0).jjtGetNumChildren()-1){s +=" .";}
                }
        } else {    s += "{"+node.jjtGetChild(0).jjtAccept(this,data);
                    s += "}_";
                    s += node.jjtGetChild(1).jjtAccept(this,data);
        }   return s;
}
```

2.2.2. Mutations avec deux passes

Ce genre de mutations, nécessite deux visiteurs, le premier va analyser le protocole et préparer les données pour le deuxième visiteur qui va appliquer la mutation proprement dite. Le premier visiteur peut être de plusieurs types, soit un visiteur qui va analyser la spécification et identifie toutes les paires (msg émis/msg reçu correspondant) et les stocker dans une liste qui sera accessible par le deuxième visiteur, soit un visiteur qui va chercher à identifier toutes les fonctions de hachage présentes dans le programme et les retourner pour le deuxième pour la suppression ou bien un visiteur qui nous repère les clés publiques utilisées dans la session de communication. Ce dernier sera particulièrement utile pour la mutation qui concerne les sessions jouées par des clés publiques identiques.

A titre d'exemple, nous donnons la fonction qui retourne les paires de messages après analyse par le visiteur.

```
public ArrayList<Pair<SimpleNode,SimpleNode>> getCouples() {
	ArrayList<Pair<SimpleNode,SimpleNode>> ret = new ArrayList<Pair<SimpleNode,SimpleNode>>();
	// pour chaque role qui a des actions
	for (String role1 : messagesDansActionsParRole.keySet()) {
		// pour chaque role qui a des gardes
		for (String role2 : messagesDansGardesParRole.keySet()) {
			// si les roles sont differents
			if (! role1.equals(role2)) {
				// on recupere les listes de noeuds de messages
				ArrayList<SimpleNode> liste1 = messagesDansActionsParRole.get(role1);
				ArrayList<SimpleNode> liste2 = messagesDansGardesParRole.get(role2);
				// on les parcours
				for (SimpleNode n1 : liste1) {
					for (SimpleNode n2 : liste2) {
						// on prend pour chaque message le nom de la fonction
						String value1 = (String) n1.jjtGetChild(0).jjtAccept(this,"");
						String value2 = (String) n2.jjtGetChild(0).jjtAccept(this,"");
						 // on regarde si les signatures sont globalement les mêmes -->
						 on cree une paire (action/envoi -> garde/reception)
						 if (! value1.equals(value2) && CANAL.contains(value1)
						 && CANAL.contains(value2) && getSignature(n1).equals(getSignature(n2))) {
							ret.add(new Pair<SimpleNode,SimpleNode>(n1,n2));
						 }
					}
				}
			}
		}
	}
	return ret;
}
```

2.2.2.1. Concaténation / Permutation

Afin d'appliquer une mutation qui concerne la concaténation / permutation, dans ce cas, nous lancerons le visiteur qui va localiser toutes les paires de messages possibles. Et puis le deuxième aura pour tâche d'appliquer la mutation sur le fichier de spécification en choisissant une seule paire à la fois, jusqu'à consommer toutes les paires (*i.e.* le deuxième visiteur sera exécuter autant de fois qu'il y en a de paires).

Les différents patterns sont :

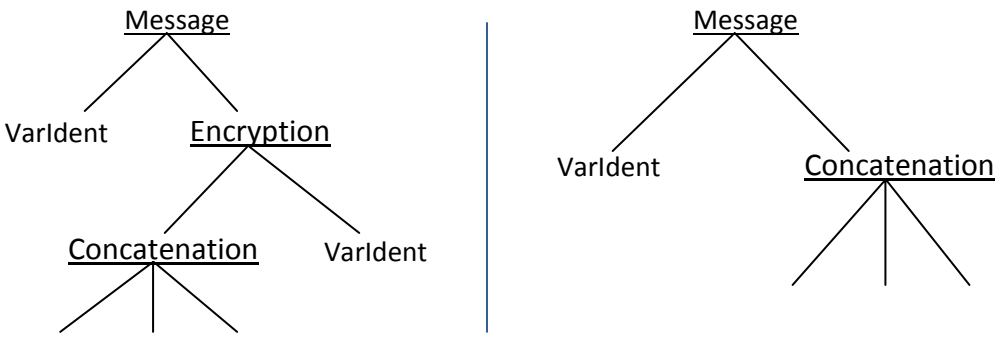


Figure 2.6 : les motifs qui correspondent à la permutation/concaténation

Afin d'éviter des éventuelles erreurs lors de l'exécution du deuxième visiteur, nous vérifierons que la paire passée au deuxième visiteur accepte cette mutation, *i.e.* le contenu du message contient une concaténation.

Le deuxième visiteur va juste s'intéresser au nœud Concaténation, dans lequel il va réaliser la permutation comme suit :

```
public Object visit(ASTConcatenation node, Object data) {
        String s = "";
        if (permut && (node.jjtGetParent() instanceof ASTEncryption
        || node.jjtGetParent() instanceof ASTMessage)) {
                permut = false;
                s += node.jjtGetChild(node.jjtGetNumChildren()-1).jjtAccept(this,data);
                for (int i=0; i < node.jjtGetNumChildren()-1; i++) {
                        s += "."+node.jjtGetChild(i).jjtAccept(this,data);
                }
        } else {
                s += node.jjtGetChild(0).jjtAccept(this,data);
                for (int i=1; i < node.jjtGetNumChildren(); i++) {
                        s += "."+node.jjtGetChild(i).jjtAccept(this,data);
                }
        }
        return s;
}
```

2.2.2.2. Suppression des fonctions de hachage

Pour assurer la bonne réalisation de cette mutation, nous utiliserons en premier lieu le visiteur qui cherche à identifier tous les noms de fonctions de hachage, et ainsi nous pourrions lancer le deuxième visiteur qui aura pour tâche de supprimer une des fonctions que son prédécesseur a identifiée. Dans la figure 2.7, on s'intéresse plus particulièrement au premier fils de "Variable_declaration" c'est-à-dire "Variable_list" dans le cas de plusieurs fonctions ou bien "VarIdent" dans le cas d'une seule fonction. Les motifs à chercher sont les suivants :

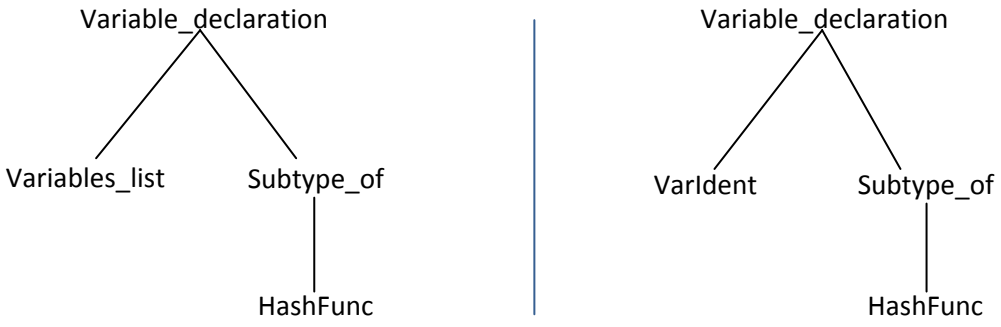


Figure 2.7 : les patterns correspondants aux fonctions de hachage

Les classes responsables de la suppression commencent par tester si le nom de la fonction rencontrée correspond à celui de la fonction à supprimer, et dans ce cas là, la fonction est évidement retirée, sinon ignorée.

Ces classes sont les suivantes :

```
public Object visit(ASTFunctionCall node, Object data) {
                String s = "";

                //fct_supp représente la fonction à supprimer

                if (fct_supp.equals(node.jjtGetChild(0).jjtAccept(this,data))){
                   for (int i=1; i < node.jjtGetNumChildren(); i++) {
                           s +=node.jjtGetChild(i).jjtAccept(this,data);
                   }
                }else{
                   s += node.jjtGetChild(0).jjtAccept(this,data)+"(";
                   for (int i=1; i < node.jjtGetNumChildren(); i++) {
                           s +=node.jjtGetChild(i).jjtAccept(this,data);
                   }
                   s += ")";
                }
                return s;
        }
```

```
public Object visit(ASTHash node, Object data) {
                String s ="";
                //fct_supp représente la fonction à supprimer
                if (fct_supp == "hash"){
                   s += node.jjtGetChild(0).jjtAccept(this,data);
                }else{
                   s +=" hash(";
                   s += node.jjtGetChild(0).jjtAccept(this,data);
                   s += ") ";
                }
                return s;
        }
```

2.2.2.3. Occultation / Substitution

Cette mutation concerne la substitution d'une partie d'un message par une autre, à titre d'exemple nous remplacerons un message comme : RCV({Na.Nb.A}_k) par RCV({Na.REMPLACE_VAR}_k), toute en veillant bien sûr à déclarer la variable REMPLACE_VAR comme par exemple de type "text" dans le protocole muté. Cette mutation aura besoin de l'exécution du visiteur qui calcule les paires de messages, et puis un deuxième visiteur va substituer une partie du message de la paire choisie par notre variable "REMPLACE_VAR".

Afin de s'assurer de la bonne validité des paires choisies, nous utiliserons une fonction de test déclarée dans le premier visiteur, avant de lancer le second. Cette fonction est la suivante

```
public boolean existsConcatenation(SimpleNode n) {
      n = (SimpleNode) n.jjtGetChild(1);
      return n instanceof ASTConcatenation
     || n instanceof ASTEncryption && n.jjtGetChild(0) instanceof ASTConcatenation;
   }
```

Nous aurons besoin aussi d'une variable qui va nous guider à bien déclarer la nouvelle variable "REMPLACE_VAR" dans les endroits voulus (c-à-d dans les rôles basiques mais pas dans les composés), ainsi, nous ajouterons la variable boolienne "autorisation" sur le nœud "Basic_role" qui nous autorise donc, à ajouter des déclarations locales dans le rôle concerné.

La déclaration de la nouvelle variable se fait dans la partie déclaration locale des agents qui participe à la session de communication ; elle est illustrée dans ce qui suit :

```
public Object visit(ASTLocal_declaration node, Object data) {
        String s = "\n          local ";
        s += node.jjtGetChild(0).jjtAccept(this,data);
        if(autorisation){ s +=","+"\n       REMPLACE_VAR : text";}
        return s;
}
```

2.2.2.4. utilisation de clés publiques identiques

Comme énoncé précédemment, dans ce modèle de mutation (qui ne concerne que les protocoles à clés publiques), nous allons exécuter les sessions de communication avec les mêmes clés publiques pour tous les agents qui y participent. Voici la classe qui récupère les clés publiques utilisées lors de l'initialisation de l'environnement d'exécution du protocole.

```
public Object visit(ASTConstant_declaration node, Object data) {
        String s ="";
        if(in_environment && node.jjtGetChild(1).toString().equals("PubKey")){
                // recuperer les clefs publiques déclarées dans le roles environment
                if(node.jjtGetChild(0).toString() == "Constants_list"){
                        for (int i=0; i < node.jjtGetChild(0).jjtGetNumChildren(); i++) {
                            clef.add(node.jjtGetChild(0).jjtGetChild(i).jjtAccept(this,data).toString());
                        }
                }else{
                        clef.add(node.jjtGetChild(0).jjtAccept(this,data).toString());
                }
        }
        s += node.jjtGetChild(0).jjtAccept(this,data);
        s += ": "+node.jjtGetChild(1).jjtAccept(this,data);
        return s;
}
```

La variable booléenne "in_environment" nous informe si on est sur le motif recherché (c-à-d dans la partie déclaration du rôle "environment"), ainsi, elle est mise à vrai dès que le visiteur traverse le nœud qui représente ce dernier (c-à-d le rôle "environment").

La classe principale du deuxième visiteur, *i.e.* celle qui va appliquer la mutation est comme suite :

```
public Object visit(ASTExpressions_list node, Object data) {
        String s ="";
        if (in_environment){
                if (liste.contains(node.jjtGetChild(0).jjtAccept(this,data))){
                        s +=liste.get(clef);
                }else{s += node.jjtGetChild(0).jjtAccept(this,data);}
                for (int i=1; i < node.jjtGetNumChildren(); i++) {
                        if (liste.contains(node.jjtGetChild(i).jjtAccept(this,data))){
                                s += ", ";
                                s +=liste.get(clef);
                        }else{
                                s += ", ";
                                s +=node.jjtGetChild(i).jjtAccept(this,data);
                        }
                }
        }else{  s += node.jjtGetChild(0).jjtAccept(this,data);
                for (int i=1; i < node.jjtGetNumChildren(); i++) {
                        s += ", ";
                        s +=node.jjtGetChild(i).jjtAccept(this,data);
                }
        } return s;
}
```

Afin de comprendre ce que fait cette classe, voici ce quand peut rencontrer dans le rôle "environment" du protocole nspk.hlpsl et son équivalant sur l'arbre syntaxique :

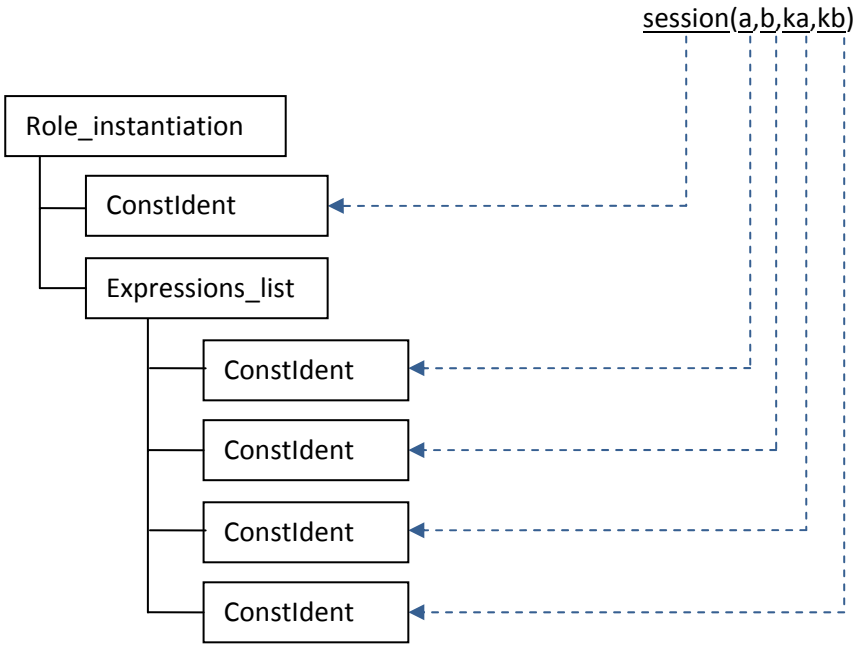


Figure 2.8 : fragment d'arbre AST représentant "session(a,b,ka,kb)"

A la fin de l'exécution du deuxième visiteur, nous aurons deux mutants qui contiendront respectivement "session(a,b,ka,ka)" et "session(a,b,kb,kb)".

Avant de clore cette section qui a présentée l'implémentation de notre modèle de fautes, nous allons donner le résultat de ce modèle appliqué au protocole donné dans le chapitre ***Etat de l'art, section 4.6*** (la mutation à appliquer est le cryptage par ***XOR***).

Résultat :

(Le mutant généré aura le nom : ***" nom d'origine"*** concaténé avec **"_"** concaténé avec ***"type de mutation appliquée"***).

```
role server ( A,B: agent, K: symmetric_key,
              H: hash_func, Snd,Rec: channel(dy))
played_by A
def=
    local  State      : nat,
           Na         : text
    init State := 0
    transition
    1.   State = 0 /\ Rec(start) =|> State' := 1
                   /\ Na' := new() /\ Snd(Na')
    2.   State = 1 /\ Rec(xor(Na,K)) =|>
                   State' := 2 /\ K' := H(K)/\
                request(A,B,auth_client,xor(Na,K))
 end role
%%%%%%%%%%%%%%%%%%%%%%%%%%%
role client ( B,A: agent,K: symmetric_key,
              H: hash_func, Snd, Rec: channel(dy))
played_by B
def=
    local  State          : nat
           Na             : text
    init State := 0
    transition
    1.   State = 0 /\ Rec(Na') =|> State' := 1
                /\ Snd(xor(Na',K)) /\ K' := H(K) /\
                witness(B,A,auth_client,xor(Na',K))
end role
```

```
role session(A,B : agent,
             K : symmetric_key,
             H: hash_func)
def=
  local St,Rt,Sl,Rl : channel(dy)
  composition
             client(B,A,K,H,St,Rt)
             /\ server(A,B,K,H,Sl,Rl)
end role
%%%%%%%%%%%%%%%%%%%%%%%%%%%
role environment()
def=
  const a,b : agent,
        k : symmetric_key,
        h : hash_func,
  auth_client:protocol_id
  intruder_knowledge = {a,b,h}
  composition
        session(a,b,k,h)
end role
%%%%%%%%%%%%%%%%%%%%%%%%%%%
goal
    authentication_on auth_client
end goal
%%%%%%%%%%%%%%%%%%%%%%%%%%%
environment()
```

3. Expérimentation

Afin d'expérimenter notre modèle de fautes, nous avons sélectionné une batterie de protocoles de sécurité sur le site officiel du projet AVISPA[7]. Cette batterie est constituée de **cinquante** protocoles considérés **sûrs** par l'outil AVISPA.

Pour entamer l'expérimentation, nous commençons par donner un exemple d'un cas de test qui permet de détecter la fameuse attaque ***man-in-the-middle***[8] sur le protocole NSPK. Ce cas de test est généré par le back-end ***"OFMC"*** de l'outil AVISPA.

Un cas de test est une suite de transitions qui permet de détecter une attaque (si elle existe) sur le protocole de sécurité analysé.

[7] http://avispa-project.org

[8] Cette attaque est illustrée dans le chapitre ***Etat de l'art, section 2***

Figure 2.9 : Cas de test du protocole NSPK ***("Attaque Man-in-The-middle")***

```
% OFMC
% Version of 2006/02/13
SUMMARY
  UNSAFE
DETAILS
      ATTACK_FOUND
PROTOCOL
      /home/avispa/web-interface-computation/
     ./tempdir/workfileIHeEiT.if
GOAL
      secrecy_of_nb
BACKEND
      OFMC
COMMENTS
STATISTICS
      parseTime: 0.00s
     searchTime: 0.03s
     visitedNodes: 10 nodes
     depth: 2 plies
ATTACK TRACE
     i --> (a,6) : start
     (a,6) --> i : {Na(1).a}_ki
     i --> (b,3) : {Na(1).a}_kb
     (b,3) --> i : {Na(1).Nb(2)}_ka
     i --> (a,6) : {Na(1).Nb(2)}_ka
     (a,6) --> i : {Nb(2)}_ki
     i --> (i,17): Nb(2)
     i --> (i,17): Nb(2)
```

Pour faciliter l'analyse des résultats, notre programme chargé d'appliquer les mutations, va classer les mutants générés suivant leurs types. Ainsi nous aurons sept classes de mutants : le cryptage par XOR (**XOR**), par EXP (**EXP**), homomorphisme (**HOMOMORPH**), même clés publiques (**PUBLIC**), substitution (**SUBSTIT**) ou permutation (**PERMUT**) d'éléments du message et suppression des fonctions de hachage (**HACH**).

3.1. Analyse des résultats

Pour des besoins de rapidité et d'efficacité, nous avons créé deux scripts qui sont donnés en annexe. Le premier a pour tâche le lancement automatique du programme de mutation sur chacun des 50 protocoles, et le deuxième, de soumettre chaque mutant généré sous l'outil AVISPA pour analyse et ranger le résultat suivant quatre catégories (**safe** pour les mutants restés sûrs, **unsafe** pour les mutants contenant des failles, **inconclusive** pour les mutants dont les vérifications ne sont pas conclusives et enfin les **mutants_incohérents** pour les mutants qui présentent des erreurs syntaxiques ou sémantiques. Nos vérifications sont réalisées par le back-end **CL-AtSe** qui prend en charge les opérateurs XOR et EXP.

3.1.1. Génération des mutants

Après l'exécution du programme de mutation sur les **50** protocoles, **1089** mutants sont générés. Et leurs nombres suivant le type de mutation est donnée dans la figure 2.9.

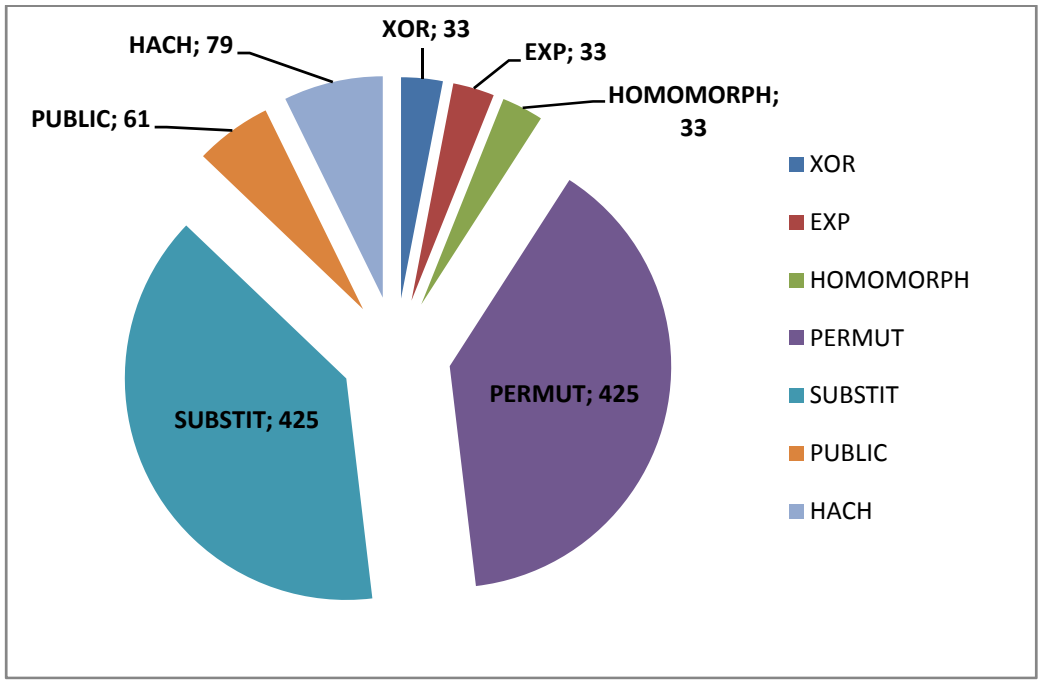


Figure 2.10 : Nombres de mutants par chaque type de mutation

Sur la figure 2.10 :

- on remarque que le nombre de mutants des types XOR, EXP et HOMOMORPH sont **égaux** et relativement **peu nombreux** (33/1089 **(3%)** pour chaque type), et cela s'explique par
    1- Les trois types s'appliquent sur les **mêmes motifs** dans l'arbre AST représentant le protocole.
    2- Ce type de mutation s'applique au **maximum** une seule fois sur chaque protocole, ce qui veut dire, que le nombre de mutants sera forcement inferieur à **50**.

    Une autre conclusion sur ce résultat, est que le nombre de protocoles utilisant le cryptage parmi les cinquante est de "**33**".
- Idem pour les types PERMUT et SUBSTIT qui s'appliquent sur des paires de messages, et dans un protocole, plusieurs échanges de messages sont effectués entre les agents, et donc plusieurs paires, ce qui explique le nombre important de mutants (425/1089 **(39%)** pour chaque type).
- Un score de 61/1089 **(5,6%)** pour les mutations qui concernent les clés publiques et un score de 79/1089 **(7,25%)** pour les mutations qui concernent les fonctions de hachage. Ces scores dépendent respectivement du nombre de protocoles qui

utilisent les clés publiques et le nombre d'agents dans le protocole en question, ou bien du nombre de protocoles qui utilise les fonctions de hachage, ainsi que leurs nombres.

3.1.2. Génération des tests

Après exécution du script de test sur les 1089 mutants, nous avons eu les résultats suivants :

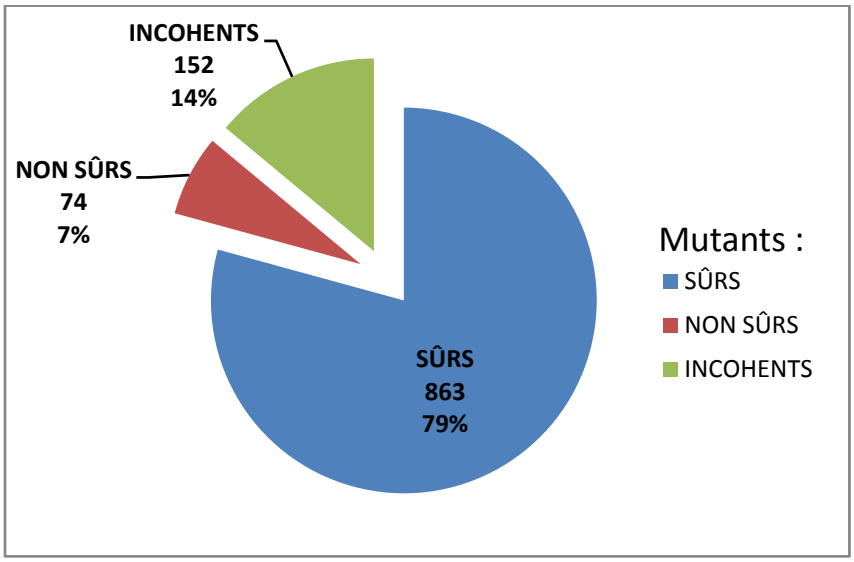
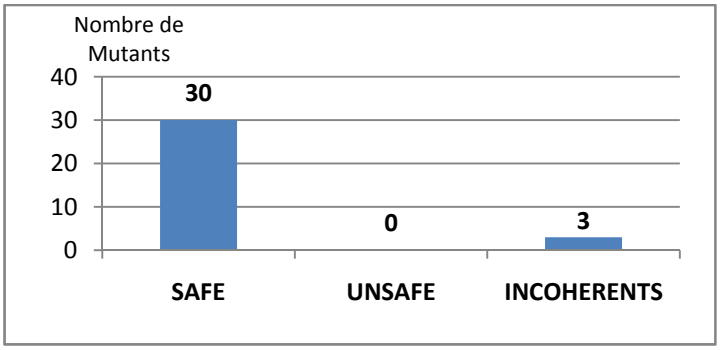


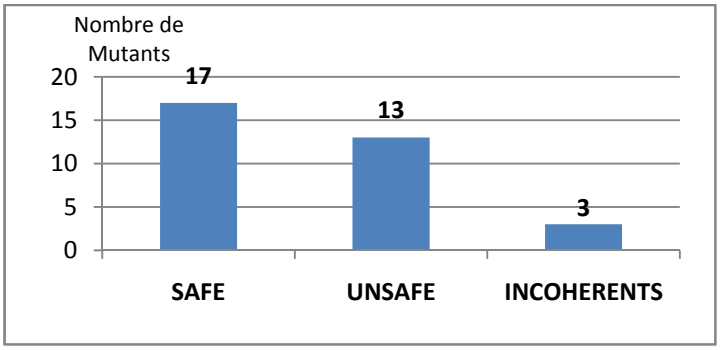
Figure 2.11 : résultats des tests d'après l'outil CL-AtSe

On remarque sur la figure 2.11 que malgré les modifications effectuées sur les spécifications de protocoles, 863/1089 mutants **(79%)** restent sans failles (sûrs), et que 74/1089 **(7%)** présentent des failles de sécurité (ce sont *cela qui nous intéressent car leurs résultats contiennent les traces d'exécutions qui permettent de détecter les attaques exploitant ces failles*). Le score des mutants incohérents **(14%)** s'explique principalement à un score élevé dans les mutations qui concernent la substitution, que nous allons développer un peu plus loin.

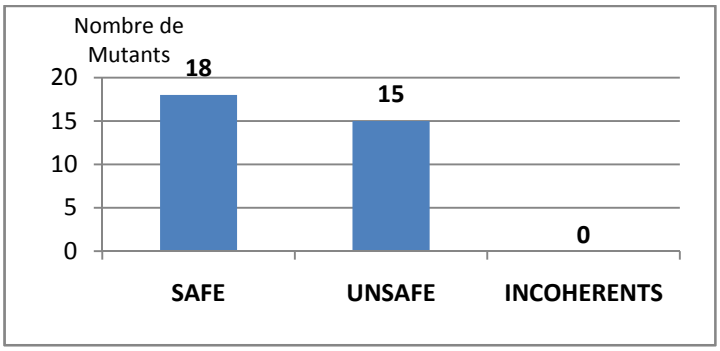
i. Génération des tests pour "EXP" (33 mutants)



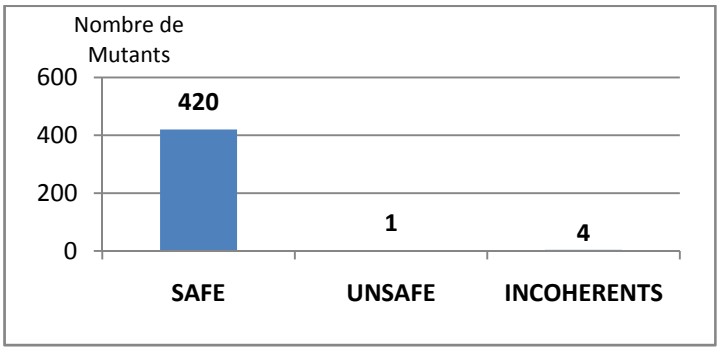
ii. **Génération des tests pour "XOR"** (33 mutants)



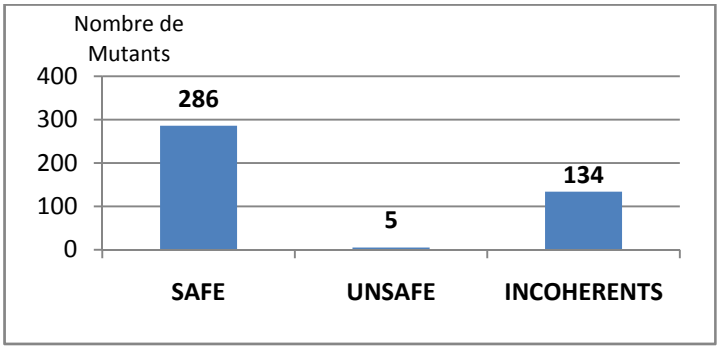
iii. **Génération des tests pour "HOMOMORPHISME"** (33 mutants)



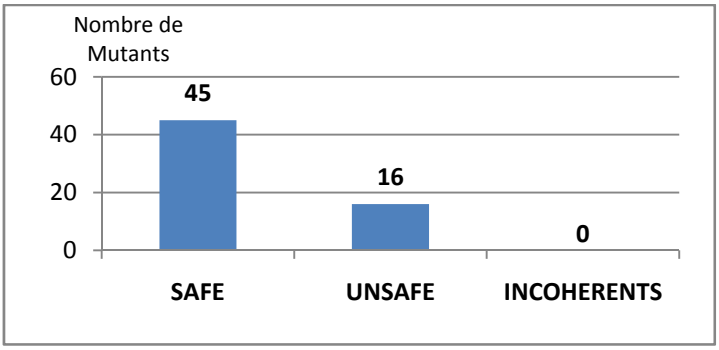
iv. **Génération des tests pour "PERMUTATION"** (425 mutants)



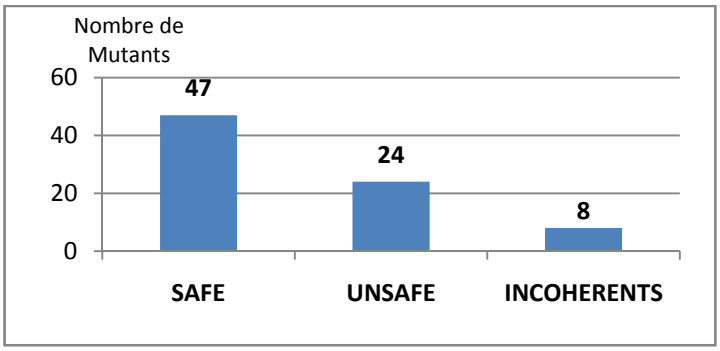
v. **Génération des tests pour "SUBSTITUTION"** (425 mutants)



vi. Génération des tests pour "CLEFS PUBLIQUES" (61 mutants)



vii. Génération des tests pour "HACHAGE" (79 mutants)



➢ **Récapitulatif des tests des 1089 mutants**

Type de mutation / Type du mutant	EXP	XOR	HOMOMORPH	PERMUT	SUBSTIT	PUBLIC	HACH	TOTAL
SAFE	30	17	18	420	286	45	47	863
UNSAFE	0	13	15	1	5	16	24	74
INCOHERENTS	3	3	0	4	134	0	8	152
TOTAL	33	33	33	425	425	61	79	1089

Figure 2.12 : Tableau récapitulatif des résultats des tests par l'outil AVISPA

Les informations essentielles à tirées de toutes ces figures est que :

- Les types de mutations qui génèrent le plus de failles sont : la mutation avec le XOR (13/33 **(39,5%)** de mutants résultants sont défaillants), l'homomorphisme **(45,5%)**, mutation avec les clés publiques **(26,2%)** et le hachage **(30,38%)**.
- La plupart des protocoles restent sans faille même après modification (après application d'une mutation).

- Le remplacement du cryptage utilisé par un protocole par un cryptage par l'opérateur **EXP** n'a pas engendré des failles de sécurité.
- Pour la mutation qui concerne le XOR, 4 mutants parmi les 17 de type safe génère une exception "**Stack OverFlow**" sur CL-AtSe, mais exécutés sans erreurs sur le back-end **"OFMC"** qui prend aussi en charge le XOR.
- Notre modèle de fautes est globalement correcte et pertinent, sauf pour la mutation par substitution, qui génère beaucoup de mutants incorrects. Ceci est dû essentiellement au fait que la plupart des protocoles sont typés. Par ailleurs ; le fait de substituer un élément dans un message qui est généralement réutilisé dans d'autre transitions, conduit forcement à des erreurs lors de la simulation du protocole.

IV. Conclusion et perspectives

Résumé

L'usage général du web et des services en ligne ont contraint les entreprises à ouvrir leurs systèmes d'information à internet, et depuis, on assiste à une explosion du nombre d'attaques sur ces systèmes. Ces attaques sont plus au moins dangereuses, et la plupart d'entre elles peuvent très bien être évitées en adoptant les bonnes pratiques comme l'utilisation et aussi la validation des protocole de sécurité avant leurs lancement sur le marché grâce aux différentes techniques de test que nous avons survolées dans le chapitre Etat de l'art et spécialement le test par mutation que nous avons expérimenté dans ce stage et qui a donné des résultats concluants dans l'ensemble.

La plupart des mutants obtenus gardent la propriété de sûreté de leurs protocoles originaux. Nos cas de tests (obtenus grâce à la catégorie des mutants "unsafe") se présentent sous forme de traces de transitions qui mènent à la découverte du secret protégé par le protocole. Un ensemble important de protocoles incohérents est généré à cause principalement de la mutation par substitution qui produit la quasi-totalité de ces mutants non désirés.

Intérêt de notre modèle de fautes

Notre modèle de fautes, comme nous l'avons vu, est implanté dans une architecture par composants, c'est-à-dire que chaque type de mutation est définit indépendamment des autres ce qui le rend plus flexible et extensible, c'est-à-dire qu'il permet de modifier ou le supprimer un de ses composants ou encore d'en ajouter un (qui va représenter un autre type de mutation) sans affecter l'architecture et le fonctionnement des autres composants. Cette qualité est assurée par le patron de conception "Visiteur" que nous avons expliqué précédemment.

Perspectives de notre application

Afin de pouvoir générer plus de tests, il faut élargir le champ d'application de notre modèle à d'autres mutations ou modification et extension de celles déjà existantes. Parmi ces extensions, on peut citer :

- ***l'affaiblissement de la garde des transitions :*** cette catégorie de fautes est reliée au prédicat ou à l'action de la transition. Aux prédicats, car nous allons supprimer des conditions dans les prédicats des gardes, et aux actions, car par cette suppression, les actions se déclencheront plus facilement et plus fréquemment.
- ***Affinement de l'analyse et l'implémentation de la mutation par substitution*** pour ne prendre en compte que les paires de messages susceptibles de générer des failles et non des erreurs lors de l'exécution du protocole. Ainsi, réduire le nombre de mutants incohérents toute en augmentant le nombre de failles à détecter.
- ***Extension de la mutation par permutation*** qui touchait aux éléments du message envoyé ou reçu à d'autre choix d'ordre de permutation au lieu d'un seul (qui concernait le dernier élément à rendre à la tête du message dans notre modèle).

Un autre point important à signaler pour les perspectives, est l'application de nos résultats sur un système réel, c'est-à-dire la concrétisation de ces tests issus du modèle de fautes à des tests exécutables. Cela revient à les implémenter sous forme d'une application qui va tourner sur le système sous test (qui utilise bien sûr un des protocoles testés ou même un autre protocole mais qui présente le même type de failles déjà présentes dans notre batterie de protocoles mutants, ce qui va permettre de la détecter). En d'autres mots, cette perspective se résume en trois points :

- Implémenter un ou plusieurs protocoles mutants représentant une faille de sécurité, et le faire tourner entre plusieurs hôtes qui communiquent.
- Implémenter les cas de tests et les faire jouer par un "intrus" afin de parvenir à connaitre le ou les secrets partagés entre lesdits hôtes.
- Une troisième étape (*qui est le but de l'implémentation*) consiste à faire jouer ces cas de tests sur d'autres protocoles non testés, et de parvenir à casser la confidentialité des messages échangés dans le cas où ce dernier présente une faille prise en compte par les cas de tests.

Pour conclure, ce projet m'a été d'un grand apport pédagogique puisqu'il m'a permis d'une part, de découvrir la sécurité informatique, et les différentes techniques de test qui existent et en expérimenter une, et d'autre part de maitriser l'outil de génération de parseurs javaCC, et de m'initier au langage de spécification de protocoles et à l'utilisation de l'outil de AVISPA pour la vérification et la validation de ces derniers.

V. Annexes

Annexe A :

Grammaire jjtree du langage HLPSL

```
/** Fichier parserHLPSL.jjt represente un parser pour le langage HLPSL utilisé par l'outil Avispa.
* Ce parser est inspiré de la grammaire donnée dans "Avispa v1.1 User Manual"
* Ce fichier decrit un analyseur syntaxique qui produit
* une representation sous forme d'arbre de syntaxe abstraite (ASA)
* d'une d'une specification en hlpsl.
* L'option MULTI permet de typer les arbres par ASTNomArbre.
* Par exemple, l'arbre annote' avec #Spec est de type ASTSpec.
* Cette classe est creee automatiquement par JJTree
* et est modifiable. Elle herite de la classe SimpleNode.
**/
options {
 MULTI=true;
 LOOKAHEAD=2;
 VISITOR=true;
}
PARSER_BEGIN(parserHLPSL)
        public class parserHLPSL {
                public static void main(String args[]) {
                        parserHLPSL parser;
                        if (args.length == 1) {
                                try {
                                        parser = new parserHLPSL(new java.io.FileInputStream(args[0]));
                                }
                                catch (java.io.FileNotFoundException e) {
                                        System.out.println("File " + args[0] + " not found.");
                                        return;
                                }
                        } else {
                                System.out.println("Usage is : java parserHLPSL inputfile");
                                return;
                        }try {
                                ASTSpec spec = parser.SpecHLPSL();
                                spec.dump(" > ");
                                System.out.println("HLPSL program parsed successfully.");
                        } catch (ParseException e) {
                                System.out.println(e.getMessage());
                                System.out.println("Encountered errors during parse.");
                        }
                }
        }
```

```
PARSER_END(parserHLPSL)
/** ---------- Regles lexicales ---------- */
SKIP : { " " | "\r" | "\t" | "\n" }
TOKEN : {
                                        < ROLE : "role" >
                                        | < PLAYED_BY : "played_by" >
                                        | < END_ROLE : "end role" >
                                        | < PAR_G : "(" >
                                        | < PAR_D : ")" >
                                        | < COMMA : "," >
                                        | < DEF : "def=" >
                                        | < LOCAL : "local" >
                                        | < OWNS : "owns" >
                                        | < CONST : "const" >
                                        | < INIT : "init" >
                                        | < AND : "/\\" >
                                        | < AFFECT : ":=" >
                                        | < ACCEPT : "accept" >
                                        | < INTRUDER_KNOWLEDGE : "intruder_knowledge" >
                                        | < EQUAL : "=" >
                                        | < ACCO_G : "{" >
                                        | < ACCO_D : "}" >
                                        | < TRANSITION : "transition" >
                                        | < POINT : "." >
                                        | < SPONTANEOUS_ACT : "--|>" >
                                        | < IMMEDIATE_REACT : "=|>" >
                                        | < NOT : "not" >
                                        | < START :"start" >
                                        | < NEW : "new()" >
                                        | < SECRET : "secret" >
                                        | < WITNESS : "witness" >
                                        | < REQUEST : "request" >
                                        | < WREQUEST : "wrequest" >
                                        | < PRIME : "'" >
                                        | < COMPOSITION : "composition" >
                                        | < POINT_VIR : ";" >
                                        | < TIRET : "_" >
                                        | < IN : "in" >
                                        | < GOAL : "goal" >
                                        | < END_GOAL : "end goal" >
                                        | < SECRECY_OF : "secrecy_of" >
                                        | < AUTH_ON : "authentication_on" >
                                        | < WEAK_AUTH : "weak_authentication_on" >
                                        | < ALWAYS : "[]" >
                                        | < SOMETIMES : "<->" >
                                        | < ONE_TIME : "(-)" >
                                        | < GLOBALLY : "[-]" >
                                        | < NEGATION : "~" >
                                        | < OR : "\\/" >
```

```
                                        | < IMPLIES : "=>" >
                                        | < DIFFERENT : "/=" >
                                        | < LESS_EQUAL : "<=" >
                                        | < DEUX_PT : ":" >
                                        | < FLECHE : "->" >
                                        | < AGENT : "agent" >
                                        | < CHANNEL_DY: "channel"([" "])*"("([" "])*"dy"([" "])*")">
                                        | < CHANNEL_OTA : "channel"([" "])*"("([" "])*"ota"([" "])*")">
                                        | < CHANNEL : "channel" >
                                        | < PUB_KEY : "public_key" >
                                        | < SYM_KEY : "symmetric_key" >
                                        | < TEXT : "text" >
                                        | < MSG : "message" >
                                        | < PROTO_ID : "protocol_id" >
                                        | < NAT : "nat" >
                                        | < BOOL : "bool" >
                                        | < HASH_FUNC : "hash_func" >
                                        | < INV : "inv" >
                                        | < HASH : "hash" >
                                        | < SET : "set" >
                                        | < DELETE : "delete" >
                                        | < CONS : "cons" >
                                        | < VAR_IDENT : <MAJ>(<MAJ>|<MIN>|<NUM>|<TIRET>)* >
                                        | < CONST_IDENT : <MIN>(<MAJ>|<MIN>|<NUM>|<TIRET>)* >
                                        | < NAT_IDENT : (<NUM>)+ >
    }
SPECIAL_TOKEN : { < LINE_COM : ("%")(~["\n","\r"])* >
                                        | < MIN : ["a"-"z"] >
                                        | < MAJ : ["A"-"Z"] >
                                        | < NUM : ["0"-"9"] >
                                        }
/** ---------------- Regles syntaxiques ---------------------- */
ASTSpec SpecHLPSL()  #Spec() : {
    /* #Spec indique que l'analyse d'une specification construit
                un ASA de classe ASTSpec qui aura autant de fils que
                de non-terminaux en partie droite de regle. */      } {
                 (Role_definition())+
                [Goal_declaration()]
                Role_instantiation()
                <EOF>{ return jjtThis; } }
void Role_definition() : {} {
                /* Roles may be either basic or compositional */
                <ROLE>
                Role_header()
                ( Basic_role() | Composition_role() )
                <END_ROLE>
          }
void Basic_role() : {} {/* Basic Role must include a player definition and generally contain a transition declaration section */
                Player()   Role_declarations()  Transition_declaration()     }
```

```
void Player() : {} {   /* Used to bind the role and the identifier of the agent playing the role */
                       <PLAYED_BY> VarIdent()      }
void Composition_role() : {} {/* Composition roles have no transition section, but rather
                         a composition section in which they call other roles. */
                       Role_declarations()
                       Composition_declaration()                }
void Role_header() : {} {                  ConstIdent() <PAR_G> [ Variables_declaration_list() ] <PAR_D>       }
void Role_declarations() : {} {
                       <DEF>
                       [Local_declaration()]
                       [Owns_declaration()]
                       [Const_declaration()]
                       [Maybe_Init_declaration()]
                       [Accept_declaration()]
                       [IKnowledge_declaration()] }
void Local_declaration() : {} { /* Declaration of local variables. */                  <LOCAL> Variables_declaration_list()   }
void Variables_declaration_list() #Variable_declaration_list : {} {Variable_declaration()(<COMMA> ariable_declaration() )*}
void Owns_declaration() : {} {             <OWNS> Variables_list()                  }
void Const_declaration() : {} {            <CONST> Constants_declaration_list() }
void Constants_declaration_list() #Constants_declaration_list(>1) : {} {
                       Constant_declaration() (<COMMA> Constant_declaration())*          }
void Maybe_Init_declaration() : {} {               <INIT> Init_declarations()     }
void Init_declarations() : {} {            Init_declaration() (<AND> Init_declaration())*    }
void Init_declaration() : {} {
                       VarIdent() <AFFECT> Expression() #AffectSansPrime(2)
                       | ConstIdent() <PAR_G> [Expressions_list()] <PAR_D> #FunctionCall(>0)
            }
void Accept_declaration() : {} {           <ACCEPT> Predicates()         }
void IKnowledge_declaration() : {} {
                       <INTRUDER_KNOWLEDGE> <EQUAL> <ACCO_G> [Expressions_list()] <ACCO_D>           }
void Transition_declaration() : {} {               <TRANSITION> (Transition())*             }
void Transition() : {} {
                       Label() <POINT>
                       Predicates()
                       (<SPONTANEOUS_ACT> Actions() #SpontaneousAction(3) |
                       <IMMEDIATE_REACT> Actions() #ImmediateReaction(3) )
                       /* spontaneous action or immediate reaction */
            }
void Label() #Label : {} {                 ConstIdent() | NatIdent()      }
void Predicates() #Predicates(>1) : {} {           Predicate() (<AND> Predicate() )*        }
void Predicate() : {} {
            <NOT> <PAR_G> Predicate() <PAR_D> #Not(1)
            | LOOKAHEAD(ConstIdent() <PAR_G>) ConstIdent() <PAR_G> [ Expressions_list() ] <PAR_D>
            #FunctionCall(>0)
            | LOOKAHEAD(VarIdent() <PAR_G>) VarIdent() <PAR_G> ( <START> #Start(1) | Expression() #Message(2) )
            <PAR_D>
            | Formula()
            }
```

```
void Actions() #Actions(>1) :  {} {   Action() (<AND> Action())*          }
void Action() #void : { } {
        VarIdent() <PRIME> <AFFECT> (<NEW> #AffectNew(1) | Expression() #Affect(2) )
        | ConstIdent() <PAR_G> [ Expressions_list() ] <PAR_D> #FunctionCall(>0)
        | VarIdent() <PAR_G> Expression() <PAR_D> #Message(2)
        | <SECRET> <PAR_G> Expression() <COMMA> ConstIdent() <COMMA> Expression() <PAR_D> #Secret(3)
        | <WITNESS> <PAR_G> Expression() <COMMA> Expression() <COMMA> ConstIdent() <COMMA> Expression()
        <PAR_D> #Witness(4)
        | <REQUEST> <PAR_G> Expression() <COMMA> Expression() <COMMA> ConstIdent() <COMMA> Expression()
        <PAR_D> #Request(4)
        | <WREQUEST> <PAR_G> Expression() <COMMA> Expression() <COMMA> ConstIdent() <COMMA> Expression()
        <PAR_D> #WRequest(4)
        }
void Composition_declaration() : {} {
                /* Definition of the composition section (for composed roles) */
                <COMPOSITION> [Compositions_list()]
        }
void Compositions_list() #Compositions_list(>0) : {Token t;} {
                Composition()
                ( t=<AND>{jjtThis.jjtSetValue(t.image);} Bracketed_par_compositions_list() #Parallel_composition
                 /* parallel */
                |t=<POINT_VIR>{jjtThis.jjtSetValue(t.image);}Bracketed_seq_compositions_list()
                 #Sequential_composition  /* sequential */
                )*
                | <PAR_G> Compositions_list() <PAR_D> #Parenthese(1) // RK parenthèses disparues ?
        }
void Composition() :{Token t;} {
                Role_instantiation()
                | t=<AND>{jjtThis.jjtSetValue(t.image);} <TIRET> <ACCO_G> Parameters_instance() <ACCO_D>
                Bracketed_compositions_list()          }
void Parameters_instance() : {Token t;} {
                t=<IN>{jjtThis.jjtSetValue(t.image);} <PAR_G> Concatenated_variables_list() <COMMA> Expression()
                <PAR_D>
        }
void Concatenated_variables_list() : {} {
                Concatenated_variables()
                | <PAR_G> Concatenated_variables() <PAR_D> #Parenthese(1) // RK parenthèses disparues ?
        }
void Concatenated_variables() #Concatenation(>1) : {} {    VarIdent() (<POINT> VarIdent())*       }
void Bracketed_par_compositions_list() : {} {
                Composition() ( <AND> Bracketed_par_compositions_list() )*
                |<PAR_G> Compositions_list() <PAR_D> #Parenthese(1) // RK parenthèses disparues ?
        }
void Bracketed_seq_compositions_list() : {} {
                Composition() ( <POINT_VIR> Bracketed_seq_compositions_list() )*
                | <PAR_G> Compositions_list() <PAR_D> #Parenthese(1) // RK parenthèses disparues ?}
void Bracketed_compositions_list() : {} {
                Composition()
                | <PAR_G> Compositions_list() <PAR_D> #Parenthese(1) // RK parenthèses disparues ?}
```

```
void Role_instantiation() : {} {                    ConstIdent() <PAR_G> [Expressions_list()] <PAR_D>          }
void Goal_declaration() : {} {              <GOAL> ( Goal_formula() )+ <END_GOAL>          }
void Goal_formula() : {} {
                    <SECRECY_OF> Constants_list() #Secrecy(1)
                    | <AUTH_ON> Constants_list() #Authenticity(1)
                    | <WEAK_AUTH> Constants_list() #WeakAuth(1)
                    | <ALWAYS> LTL_unary_formula() #Always(1)
          }
void LTL_unary_formula() : {} {
                    LTL_unary_predicate()
                    | <SOMETIMES> LTL_unary_formula() #Sometimes(1)
                    | <ONE_TIME> LTL_unary_formula() #OneTime(1)
                    | <GLOBALLY> LTL_unary_formula() #Globally(1)
                    | <NEGATION> LTL_unary_formula() #Negation(1)
                    | <PAR_G> LTL_formula() <PAR_D> #Parenthese(1)
          }
void LTL_formula() : {} {
                    LTL_predicate() [LTL_formula2()]
                    | <SOMETIMES> LTL_unary_formula() [LTL_formula2()] #Sometimes(>0)
                    | <ONE_TIME> LTL_unary_formula() [LTL_formula2()] #OneTime(>0)
                    | <GLOBALLY> LTL_unary_formula() [LTL_formula2()] #Globally(>0)
                    | <NEGATION> LTL_unary_formula() [LTL_formula2()] #Negation(>0)
                    | <PAR_G> LTL_formula() <PAR_D> [LTL_formula2()] #Parenthese(>0)
          }
void LTL_formula2() : {/* cette règle est créée pour éliminer la récursivité à gauche de la règle précédente */} {
                    <AND> LTL_formula() [LTL_formula2()] #And(>0)
                    | <OR> LTL_formula() [LTL_formula2()] #Or(>0)
                    | <IMPLIES> LTL_formula() [LTL_formula2()] #Implies(>0)  }
void LTL_unary_predicate() : {} {
                    ConstIdent() <PAR_G> [Expressions_list()] <PAR_D> #FunctionCall(>0)
                    | <IN> <PAR_G> Expression() <COMMA> VarConst() <PAR_D> #In(2)
                    | <NOT> <PAR_G> LTL_predicate() <PAR_D> #Not(1)        }
void LTL_predicate() : {} {
                    LTL_unary_predicate()
                    | Expression() (<EQUAL> Expression() #Equal(2)
                    | <LESS_EQUAL> Expression() #LessEqual(2)
                    | <DIFFERENT> Expression() #Difference(2) )      }
void Variable_declaration() : {} { Variables_list() <DEUX_PT> Type_of()          }

void Variables_list() #Variables_list(>1) : {} {       VarIdent() (<COMMA> VarIdent())*     }
void Type_of() #Type_of(>1): {} {          Subtype_of() ( <FLECHE> Subtype_of() )*}
void Subtype_of() #Subtype_of : {} {
                    ( Simple_type()
                    | <PAR_G> Subtype_of() <PAR_D> #Parenthese(1)
                    | <ACCO_G>  Subtype_of() <ACCO_D>  <TIRET> Bracketed_subtype_of() #Encryption(2)
                    | <INV> <PAR_G> Subtype_of() <PAR_D> #Inv(1)
                    | <HASH> <PAR_G> Subtype_of() <PAR_D> #Hash(1) )
                    ( <POINT> Subtype_of() #Point(1) |  <SET> #SetLiteral )*
          }
```

```
void Constant_declaration() : {} {          Constants_list() <DEUX_PT> Simple_type_of()    }
void Constants_list() #Constants_list(>1) : {} {      ConstIdent() (<COMMA> ConstIdent())*            }
void Simple_type_of() #Simple_typeof(>1): {} {    Simple_subtype_of() ( <FLECHE> Simple_subtype_of() )*   }
void Simple_subtype_of() #void : {} {
                        Simple_type()
                        | <PAR_G> Simple_type_of() <PAR_D> #Parenthese(1)
            }
void Simple_type() #void : {Token t;} {
                        t=<AGENT>{ jjtThis.jjtSetValue(t.image); } #Agent
                        | t=<CHANNEL>{ jjtThis.jjtSetValue(t.image); } #Channel
                        | t=<CHANNEL_DY>{ jjtThis.jjtSetValue(t.image); } #ChannelDY
                        | t=<CHANNEL_OTA>{ jjtThis.jjtSetValue(t.image); } #ChannelOTA
                        | t=<PUB_KEY>{ jjtThis.jjtSetValue(t.image); } #PubKey
                        | t=<SYM_KEY>{ jjtThis.jjtSetValue(t.image); } #SymKey
                        | t=<TEXT>{ jjtThis.jjtSetValue(t.image); } #Text                /* used for nonces */
                        | t=<MSG>{ jjtThis.jjtSetValue(t.image); }           #Msg                    /* generic type */
                        | t=<PROTO_ID>{ jjtThis.jjtSetValue(t.image); } #ProtoId    /* kind of label */
                        | t=<NAT>{ jjtThis.jjtSetValue(t.image); } #Nat
                        | t=<BOOL>{ jjtThis.jjtSetValue(t.image); } #Bool
                        | t=<HASH_FUNC>{ jjtThis.jjtSetValue(t.image); } #HashFunc  /* hash function */
                        | <ACCO_G> Constants_or_nat_list() <ACCO_D> #Enumeration(1) /* enumeration */
            }
void Constants_or_nat_list() #ConstantsOrNatList(>1) : {} {
                        (ConstIdent() | NatIdent()) ( <COMMA> (ConstIdent() | NatIdent()) )*
            }
void Bracketed_subtype_of() : {} {
                        Simple_type()
                        | <INV> Subtype_of() <PAR_D> #Inv(1)
                        | <PAR_G> Subtype_of() <PAR_D> #Parenthese(1)
            }
void Formula() #void : {} {
                        Expression() (<EQUAL> Expression() #Equal(2)
                        | <LESS_EQUAL> Expression() #LessEqual(2) )
                        /* Inclusion test: in(Elt,Set) */
                        | <IN> <PAR_G> Expression() <COMMA> Expression() <PAR_D> #In(2)
                        /* Syntactic sugar for inequality: */
                        | Expression() <DIFFERENT> Expression() #Different(2)
                        | <PAR_G> Formula() <PAR_D> #Parenthese(1)
            }
void Expressions_list() #Expressions_list(>1) : {} {              Expression() (<COMMA> Expression())*            }
void Expression() #Concatenation(>1) : {} {                       Concat() ( <POINT> Concat() )*          }
void Concat() #void : {} {
                        <PAR_G> Expression() <PAR_D> #Parenthese(1)
                        /* Function application: */
                        | LOOKAHEAD(VarConst() <PAR_G>) VarConst() <PAR_G> Expressions_list() <PAR_D> #FunctionCall(2)
                        /* Inverse of a public/private key: */
                        | <INV> <PAR_G> Expression() <PAR_D> #Inv(1)
                        /* Insertion of an element in a set: cons(Elt,Set) */
                        | <CONS> <PAR_G> Expression() <COMMA> Expression() <PAR_D> #Cons(2)
```

```
                    /* Deletion of an element in a set: delete(Elt,Set) */
                    | <DELETE> <PAR_G> Expression() <COMMA> Expression() <PAR_D> #Delete(2)
                    /* New value of a variable: */
                    | VarIdent() [ <PRIME> #Prime(1) ]
                    /* Old value of a variable: */
                    | ConstIdent()
                    /* Valeur */
                    | NatIdent()
                    /* Encryption: {Na'.A}_(Ka.Kb') */
                    | LOOKAHEAD(<ACCO_G> Expression() <ACCO_D> <TIRET>) <ACCO_G> Expression() <ACCO_D>
                    <TIRET> Bracketed_expression()  #Encryption(2)
                    /* Set: */
                    | <ACCO_G> Expressions_list() <ACCO_D> #Set(1)
          }
void Bracketed_expression() #void : {} {
                    <INV> <PAR_G> Expression() <PAR_D> #Inv(1)
                    | LOOKAHEAD(VarConst() <PAR_G>) VarConst() <PAR_G> Expressions_list() <PAR_D> #FunctionCall(2)
                    | ConstIdent()
                    | NatIdent()
                    | VarIdent() [ <PRIME> #Prime(1) ]
                    | <PAR_G> Expression() <PAR_D>  #Parenthese(1)
          }
void VarConst() : {} {          VarIdent() | ConstIdent() }
void VarIdent() : { Token t; } {t = <VAR_IDENT> { jjtThis.jjtSetValue(t.image); }  }
void ConstIdent() : { Token t; } {t = <CONST_IDENT> { jjtThis.jjtSetValue(t.image); } }
void NatIdent() : { Token t; } { t = <NAT_IDENT> { jjtThis.jjtSetValue(t.image); } }
```

Annexe B :

Script de génération des mutants

```
#!/bin/sh
# Fichier "generer.sh"

echo Analyse des protocoles et application des mutations en cours...
echo Veuillez patienter...

for element in *.hlpsl
        # Lancement de l'application de mutation sur chaque ficher de spécification pour la génération des mutants
        do java mutationHLPSL "$element"
done

echo génération des mutants terminée \!

# Copie du script de test dans chaque dossier représentant un type de mutation
cp tester.sh mutants\/EXP\/tester.sh
cp tester.sh mutants\/XOR\/tester.sh
cp tester.sh mutants\/HACH\/tester.sh
cp tester.sh mutants\/HOMOMORPH\/tester.sh
cp tester.sh mutants\/PERMUT\/tester.sh
cp tester.sh mutants\/PUBLIC\/tester.sh
cp tester.sh mutants\/SUBSTIT\/tester.sh

echo Les protocoles mutés sont créés et rangés par \catégorie dans le dossier \"mutants\"
echo "==>"pour les tester, veuillez executer le scrip tester.sh depuis chaque dossier dans $PWD\/mutants
```

Annexe C :

Script de génération des tests par CL-AtSe

```
#!/bin/sh
# Fichier "tester.sh"
if test ! -d test_results
        then mkdir test_results
fi
echo Analyse des protocoles par l\'outil AVISPA en cours...
echo Veuillez patienter...
for element in *.hlpsl

        do echo Analyse de "$element"
        nom=$(echo "$element" | cut -f1 -d '.')
        avispa "$element" --cl-atse > $PWD\/test_results\/test_"$nom".log

        fichier_a_deplacer=test_"$nom".log

        cd test_results

        grep -q "ERROR" "$fichier_a_deplacer"
        if [ $? -eq 0 ]
        then
                if test ! -d mutants_incohérents
                then
                        mkdir mutants_incohérents
                        mv $fichier_a_deplacer $PWD\/mutants_incohérents\/$fichier_a_deplacer.err
                else
                        mv $fichier_a_deplacer $PWD\/mutants_incohérents\/$fichier_a_deplacer.err
                fi
        fi

        grep -q "UNSAFE" "$fichier_a_deplacer" 2> /dev/null
        if [ $? -eq 0 ]
        then
                if test ! -d unsafe
                then
                        mkdir unsafe
                        mv $fichier_a_deplacer $PWD\/unsafe\/$fichier_a_deplacer
                else
                        mv $fichier_a_deplacer $PWD\/unsafe\/$fichier_a_deplacer
                fi
        fi

        grep -q "SAFE" "$fichier_a_deplacer" 2> /dev/null
        if [ $? -eq 0 ]
        then
                if test ! -d safe
                then
                        mkdir safe
                        mv $fichier_a_deplacer $PWD\/safe\/$fichier_a_deplacer
                else
                        mv $fichier_a_deplacer $PWD\/safe\/$fichier_a_deplacer
                fi
        fi
```

```
		grep -q "INCONCLUSIVE" "$fichier_a_deplacer" 2> /dev/null
		if [ $? -eq 0 ]
		then
			if test ! -d inconclusive
			then
				mkdir inconclusive
				mv $fichier_a_deplacer $PWD\/inconclusive\/$fichier_a_deplacer
			else
				mv $fichier_a_deplacer $PWD\/inconclusive\/$fichier_a_deplacer
			fi
		fi

		cd ..
done

echo Analyse terminée \!
echo Les résultats des tests sont dans \:  $PWD\/test_results \:

if test  -d $PWD\/test_results\/mutants_incohérents
then
		echo "	"- Les mutants incohérents sont dans \: $PWD\/test_results\/mutants_incohérents
fi

if test  -d $PWD\/test_results\/safe
then
		echo "	"- Les mutants restés sans failles sont dans \: $PWD\/test_results\/safe
fi

if test  -d $PWD\/test_results\/unsafe
then
		echo "	"- Les mutants avec failles sont dans \: $PWD\/test_results\/unsafe
fi

if test  -d $PWD\/test_results\/inconclusive
then
		echo "	"- Les tests inconclusifs sont dans \: $PWD\/test_results\/inconclusive
fi
```

Bibliographie

[01] Solange Ghernaouti-Hélie « Sécurité Informatique et Réseaux » DUNOD 2006

[02] Véronique Cortier, Stéphanie Delaune et Pascal Lafourcade, *« A Survey of Algebraic Properties Used in Cryptographic Protocols ».*

[03] J. Bull and D. J. Otway. The authentication protocol. Tech. Rep. DRA/CIS3/PROJ/CORBA/SC/1/CSM/436-04/03, Defence Research Agency, 1997.

[04] J. Clark and J. Jacob. A survey of authentication protocol literature. http://www.cs.york.ac.uk/~jac/papers/drareviewps.ps, 1997.

[05] Andrew R.Baker, « Snort IDS and IPS ToolKit », Syngress edition, 2007

[06] S*upport de cour du cabinet Hervé Schauer,* « Introduction à la cryptographie »

[07] http://fr.wikipedia.org

[08] Abdesselam LAKEHAL, Critères de Couverture Structurelle pour les Programmes Lustre, thèse de doctorat, université Joseph Fourier, 2006

[09] IEEE standard for software test documentation, 1998

[10] http://www.loria.fr/news/bloc3/resultats-scientifiques/avispa/

[11] http://www.avispa-project.org/

[12] AVISPA v1.1 User Manual, The AVISPA Team. June 30, 2006

[13] AVISPA – HLPSL Tutorial, A Beginner's Guide to Modelling and Analysing Internet Security Protocols; The AVISPA team

[14] David Basin, Sebastian Modersheim, and Luca Vigano - An On-The-Fly Model-Checker for Security Protocol Analysis - Department of Computer Science, ETH Zurich

[15] Alexandro Armando, Luca Compagna – SATMC a SAT-based Model Checker for Security Protocol – AI lab, University of Genova, Italy

[16] Yohan Boichut – Approximations pour la vérification automatique de protocoles de sécurité – UFR Sciences ET Techniques, Université de Franche-Comté, 2006.

[17] M. Rusinowitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In 14th EETH Computer Security Foundations Workshop, pages 174 – 190, Washington – Brussels – Tokyo, June 2001. IEEE

[18] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. 8th ACM Conference on Computer and Communications Security (CCS'01)*. ACM Press, 2001.

[19] H.Yoon, B.Choi, and J. O. Jeon, "Mutation-Based Inter-Class Testing," in Proceedings of the 5th Asia Pacific Software Engineering Conference (APSEC'98), Taipei, Taiwan,2-4 December 1998, p.174

[20] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. Masiero, "Mutation Analysis Testing for Finite State Machines," in *Proceedings of the 5th International Symposium on Software Reliability Engineering*, Monterey, California, 6-9 November 1994, pp. 220–229.

[21] M. R. Woodward and K. Halewood, "From Weak to Strong, Dead or Alive? an Analysis of Some Mutation testing Issues," in *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)*. Banff Albert, Canada: IEEE Computer Society, July 1988, pp. 152–158.

[22] cours de compilation de M. Hamdani, professeur au département informatique et électronique de l'université UMMTO (Algérie)

[23] Jérôme Voinot, http://lifc.univ-fcomte.fr/~voinot

[24] http://fr.wikipedia.org/wiki/Grammaire_non_contextuelle

[25] Guoqiang Shu et David Lee ''Message Confidentiality Testing of Security Protocols – Passive Monitoring and Active Checking'', Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA

[26] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, April 1978

[27] Yue Jia and Mark Harman, An Analysis and Survey of the Development of Mutation Testing. Technical Report TR-09-06

[28] V. Viennet, M. Nagot, J. Pistoli, C. Delima " Etude comparative d'outils d'injection de fautes ", ISTIA, Master Pro 2007.

[29] A. J. Offutt, "The Coupling Effect: Fact or Fiction," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, December 1989.

[30] A. J. Offutt, "Investigations of the Software Testing Coupling Effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, January 1992.

[31] http://www.mutationtest.net

[32] Matthieu CARLIER, Test automatique de propriétés dans un atelier de développement de logiciels sûrs. Thèse de doctorat, Conservatoire National des Arts et Métiers, 2009

[33] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, “Integration Testing Using Interface Mutation,” in *Proceedings of the seventh International Symposium on Software Reliability Engineering*, White Plains, New York, November 1996.

[34] A. S. Gopal and T. A. Budd, “Program Testing by Specification Mutation,” University of Arizona, Tucson, Arizona, Technical Report TR 83-17, 1983.

[35] T. A. Budd and A. S. Gopal, “Program Testing by Specification Mutation,” Computer Languages, vol. 10, no. 1, pp. 63–73, 1985.

[36] Jacques Farré, Techniques de compilation, Introduction à JavaCC, Université de Nice

[37] http://en.wikipedia.org/wiki/Visitor_pattern

[38] http://fr.wikibooks.org/wiki/Patrons_de_conception/Visiteur

[39] The Visitor Family of Design Patterns by Robert C. Martin - a rough chapter from *The Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall. http://objectmentor.com/resources/articles/visitor.pdf

[40] Bertrand Meyer and Karine Arnout, Article "Componentization: the Visitor Example", *Computer* (IEEE), vol. 39, no. 7, July 2006, pages 23-30. http://se.ethz.ch/~meyer/publications/computer/visitor.pdf

[41] Peter Buchlovsky et Hayo Thielecke, ‘’ A Type-theoretic Reconstruction of the Visitor Pattern’’, 2005. http://www.cs.bham.ac.uk/~hxt/research/mfps-visitors.pdf

[42] Jens Palsberg et C. Barry Jay, ‘’The Essence of the Visitor Pattern’’, 2007

[43] Carine COURBIS, Thèse ‘’ Contribution à la programmation générative, Application dans le générateur smarttools : technologies xml, programmation par aspects et composants ’’. Université de Nice Sophia-Antipolis, décembre 2002.

[44] A. Je_erson Offutt et Roland H. Untch, "Mutation 2000: Uniting the Orthogonal"