Établir la non-traçabilité de protocoles cryptographiques avec Proverif

Benjamin Bordais
École Normale Supéreure de Rennes
Département Informatique et Télécommunication

sous la supervision de
Stéphanie Delaune
Chercheur CNRS, Équipe EMSEC, Irisa Rennes

10 juillet 2017

Résumé

La sécurité des échanges quotidiens, que l'on effectue via des outils de communication sans fils, est primordiale. C'est pourquoi l'étude des protocoles cryptographiques régissant ces échanges est, à présent, incontournable. Celle-ci requiert bien souvent une automatisation du processus permettant une vérification formelle de certaines propriétés sur un protocole donné. Ce stage vise à utiliser l'outil de vérification Proverif afin de vérifier automatiquement la propriété de non-traçabilité.

Mots-clés : Sécurité ; Protocole Cryptographique ; Vérification Formelle ; Proverif ; Non-traçabilité.

Table des matières

Introduction

Nous vivons dans une société qui s'appuie essentiellement sur notre faculté à communiquer de façon sécurisée. Cela est rendu possible par l'utilisation de protocoles cryptographiques. Il s'agit d'une façon de déterminer le rôle de chaque agent communiquant pouvant utiliser les primitives cryptographiques (telles que des fonctions de chiffrement ou de déchiffrement) à leur disposition. Cependant, ces protocoles comportent généralement des failles de sécurité. Il s'ensuit alors une succession de découvertes de nouveaux moyens destinés à mettre à défaut un protocole ou bien à le renforcer.

Pour pouvoir disposer de protocoles plus sûrs, on peut opter pour une étude automatisée des protocoles cryptographiques, via l'utilisation de méthodes formelles. De par leur formalisme rigoureux, elles permettent une analyse soigneuse qui apporte une assurance certaine sur le niveau de sécurité du protocole étudié. Par exemple, le modèle symbolique offre de nombreuses possibilités puisqu'il permet de modéliser la majorité des propriétés de sécurité liées à la vie privée. Une telle approche a pu permettre de mettre au jour un défaut dans un protocole qu'utilisait Google Apps [1].

Cependant, la majorité des résultats les plus concluants se base sur la vérification qu'aucun événement indésirable n'est atteignable. C'est, par exemple, le cas pour les notions de secret ou d'authentification. *A contrario*, les propriétés liées à la protection de la vie privée, comme la non-traçabilité, sont plutôt basées sur la notion d'équivalence. Toutefois, vérifier une équivalence est, en général, indécidable notamment lorsque le nombre de sessions considérées n'est pas borné [4].

Pour se défaire du problème d'indécidabilité, l'idée d'utiliser des procédures de semi-décision (c'est-à-dire qui ne marchent pas à chaque fois) a émergé. Ainsi, un outil de vérification comme Proverif s'est révélé assez efficace pour établir un certain nombre de propriétés de sécurité, bien qu'il n'ait aucune garantie de terminaison ou de complétude. Malgré ses bonnes performances, Proverif n'est pas capable d'établir directement qu'un protocole assure la non-traçabilité car la notion d'équivalence qu'il utilise est trop forte par rapport à celle utilisée dans la définition de la non-traçabilité.

Mon stage s'inscrit dans ce cadre. Il s'appuie sur la thèse de Lucca Hirschi [5] qui a pu établir une condition suffisante garantissant qu'un protocole assurait la non-traçabilité. Celle-ci se divise en deux propriétés plus simples : *Well-Authentication* et *Frame Opacity*. *Well-Authentication* est une propriété d'atteignabilité et est donc assez naturelle à encoder en Proverif. Dans sa thèse, Lucca Hirschi a également testé un codage Proverif concernant la propriété de *Frame Opacity* sur des protocoles donnés. Cependant, ses résultats ne sont pas entièrement satisfaisants. L'objet de mon stage consistait à tester un autre codage Proverif et à évaluer son efficacité.

La section 1 se propose de présenter le formalisme nécessaire à la modélisation de protocoles dont il sera question par la suite. On abordera, dans la section 2, la définition précise des propriétés qui nous intéressent afin de comprendre le codage proposé, destiné à vérifier ces propriétés. La section 3 abordera la mise en place effective du codage de *Frame Opacity* et l'automatisation de la transformation des protocoles. Enfin, la section 4 sera consacrée à la démarche de validation employée pour vérifier l'exactitude et la performance de l'implémentation mise en place.

1 Préliminaires : Modéliser un protocole cryptographique

Dans cette partie, on introduit le modèle générique de protocoles cryptographiques qui sera utilisé tout au long de ce rapport.

1.1 Cadre d'étude

Pour représenter des protocoles, on peut utiliser différentes approches. L'approche computationnelle, par exemple, considère que l'attaquant peut faire ce qu'il veut, y compris essayer de casser les primitives cryptographiques tant que cela s'effectue en un temps polynomial. Ainsi, les messages sont vus comme des suites de bytes et les agents comme des machines de Turing probabilistes s'exécutant en temps polynomial. Cependant, la représentation des protocoles cryptographiques utilisée durant ce stage repose sur une autre approche : l'approche symbolique. Celle-ci suppose l'existence de primitives cryptographiques idéales. Ainsi, on peut supposer l'existence d'une fonction de chiffrement *enc* et d'une fonction de déchiffrement *dec* qui vérifient : $dec(enc(m,k),k) = m$. Cela signifie que l'on ne peut récupérer le message encrypté m que si l'on dispose de la clé k, on ne peut casser cette primitive cryptographique. Au sein de ce modèle, les capacités de l'attaquant sont entièrement déterminées par les propriétés algébriques des primitives cryptographiques utilisées dans le protocole (comme, par exemple, celle définie plus haut pour le déchiffrement). Toutefois, l'attaquant peut faire tout ce qu'il souhaite sur le réseau : il peut, par exemple, intercepter toutes les communications, ou encore injecter, à tout instant, le message de son choix sur le réseau. Cette modélisation permet une représentation abstraite des protocoles. Celle-ci est développée par la suite au sein d'un modèle qui découle du π-calcul appliqué [7].

1.2 Algèbre de termes

Les messages échangés au cours d'un protocole cryptographique sont modélisés par des termes dont on définit une algèbre.

On suppose l'existence d'un ensemble infini $\mathcal{N}$ de noms qui sont utilisés pour représenter des clés ou des nonces. Un nonce est un nombre frais, généré aléatoirement à chaque nouvelle session. Il est souvent utilisé pour représenter un agent en particulier. D'autres part, on suppose l'existence de deux ensembles infinis disjoints $\mathcal{X}$ et $\mathcal{W}$ de variables. Les variables de $\mathcal{X}$ représentent les parties inconnues des messages attendus par les différents agents, tandis que les variables de $\mathcal{W}$ stockent les connaissances de l'attaquant.

De plus, on considère une signature Σ qui est un ensemble infini de symboles de fonction ainsi que leur arité. Les symboles de fonction sont séparés en constructeurs (dont l'ensemble est noté Σ_c) et destructeurs (dont l'ensemble est noté Σ_d). On a alors $\Sigma = \Sigma_c \sqcup \Sigma_d$. Les constructeurs modélisent les fonctions totales pouvant vérifier certaines propriétés. Typiquement, le symbole de fonction *enc* évoqué plus haut, qui représente une fonction de chiffrement, est un constructeur. De leur côté, les destructeurs représentent des opérations qui peuvent échouer. Ainsi, le symbole de fonction *dec* qui représente une fonction de déchiffrement est un destructeur.

Exemple 1 *Considérons la signature :*

$$\Sigma = \{enc, dec, (), \pi_1, \pi_2, \oplus, 0, eq\}$$

Les symboles enc et dec, d'arité 2, représentent des fonctions de chiffrement et de déchiffrement. Le symbole $()$, d'arité 2, représente la fonction d'appariement tandis que π_i représente une fonction de

projection : $\pi_i(x_1, x_2) = x_i$ pour $i = 1, 2$. Le symbole, d'arité 2, $\oplus$ et la constante (ou symbole d'arité zéro) 0 sont utilisés pour modéliser le ou exclusif. Enfin, eq fait référence au test d'égalité d'arité 2. On a alors $\Sigma_c = \{enc, (), \oplus, 0\}$ et $\Sigma_d = \{dec, \pi_1, \pi_2, eq\}$.

Étant donné un ensemble $\mathcal{A}$ et une signature Σ, on note $\mathcal{T}(\Sigma, \mathcal{A})$ l'ensemble des éléments construits à partir de $\mathcal{A}$ en appliquant les symboles de fonction de Σ. On appelle alors termes les éléments de $\mathcal{T}(\Sigma, \mathcal{N} \cup \mathcal{X})$. Étant donné un terme t, on note par $var(t)$ l'ensemble des variables y apparaissant. On dit qu'un terme est clos si $var(t) = \emptyset$. Un message est un terme clos dans lequel aucun destructeur n'apparaît. Les messages sont les seuls termes à pouvoir être échangés sur le réseau.

1.2.1 Théorie équationnelle

Pour modéliser certaines propriétés de primitives cryptographiques, on utilise une théorie équationnelle. On considère tout d'abord un ensemble E d'équations sur $\mathcal{T}(\Sigma_c, \mathcal{X})$ (on ne travaille qu'avec des constructeurs). On peut alors définir une congruence $=_E$ sur $\mathcal{T}(\Sigma_c, \mathcal{N} \cup \mathcal{X})$ à partir de E, stable par substitution, application de symboles de fonction de Σ_c et renommage bijectif.

Exemple 2 *Pour modéliser les propriétés algébriques du ou exclusif, on peut utiliser la théorie équationnelle E engendrée par le système d'équations :*

$$x \oplus 0 = x \qquad (x \oplus y) \oplus z = x \oplus (y \oplus z)$$
$$x \oplus x = 0 \qquad (x \oplus y) = (y \oplus x)$$

Dans ce cas, on a $enc(a \oplus 0, k \oplus k') =_E enc(a, k' \oplus k)$. Cela signifie que les deux termes $enc(a \oplus 0, k \oplus k')$ et $enc(a, k' \oplus k)$ représentent la même information, et qu'ainsi aucun des agents ne peut les distinguer (y compris l'attaquant).

Au sein de la théorie équationnelle, les destructeurs ne sont pas pris en compte, car on ne tient pas compte d'un possible échec dans le calcul d'un terme.

1.2.2 Système de réécriture

Le calcul d'un terme peut aboutir à un message ou à un échec. Cela est modélisé à l'aide d'un système de réécriture. Un tel système décrit comment les différents destructeurs influent sur leurs arguments. Il s'agit d'un ensemble, ordonné, de règles de la forme $g(u_1, u_2, ..., u_n) \to u$ avec $g \in \Sigma_d$ et $u, u_1, u_2, ..., u_n \in \mathcal{T}(\Sigma_c, \mathcal{X})$. Ainsi, chaque règle ne définit l'action que d'un seul destructeur à la fois. L'objectif étant d'exécuter un terme clos en appliquant différentes règles de réécriture jusqu'à obtenir un message (terme clos sans destructeur) ou un échec (ce qui se produit lorsqu'aucune règle de réécriture n'est applicable). Lorsque plusieurs règles de réécriture peuvent être appliquées à un même terme, on applique celle apparaissant en premier dans le système de réécriture considéré. Lorsqu'un terme clos t peut être exécuté en un terme t', on note $t \rightsquigarrow t'$. Étant donné un système de réécriture $\mathcal{R}$, et un système d'équation E associé à une théorie équationnelle, on notera par la suite $\Downarrow$ la clôture réflexive et transitive des relations $\rightsquigarrow$ et $=_E$. Quand il n'existe pas de message u tel que $t \Downarrow u$, on notera $t \not\Downarrow$.

Exemple 3 *On pourrait associer à la signature Σ défini à l'exemple 1, le système de réécriture :*

$$dec(enc(m,k),k) \to m \qquad eq(x,x) \to ok$$
$$\pi_1(x_1,x_2) \to x_1 \qquad \pi_2(x_1,x_2) \to x_2$$

Si l'on considère le système d'équation de l'exemple 2, on a $dec(enc(a,b\oplus 0),b) \Downarrow a$ *car* $b\oplus 0 =_E b$. *En revanche,* $dec(enc(a,a\oplus b),b) \not\Downarrow$.

Ordonner les règles de réécriture permet de définir la relation d'inégalité (de symbole *neq*, d'arité 2) indépendamment de la relation d'égalité à l'aide du système de réécriture :

$$neq(x,x) \to no \qquad neq(x,y) \to yes$$

1.2.3 Connaissance de l'attaquant

La connaissance de l'attaquant est l'ensemble des messages qu'il a appris en interceptant des échanges entre différents agents. Celle-ci est organisée en *frame*. Plus formellement, une *frame* est une substitution de $\mathcal{W}$ (défini plus haut) vers l'ensemble des messages. L'attaquant peut alors créer ses propres messages en utilisant les fonctions à sa disposition. Il ne peut utiliser que les symboles de fonction dits publiques (dont l'ensemble est noté Σ_{pub}). L'ensemble des symboles de fonction privés est noté Σ_{priv}. Ces deux ensembles vérifient $\Sigma = \Sigma_{pub} \sqcup \Sigma_{priv}$.

Un calcul effectué par l'attaquant est appelé une recette qui, formellement, est un terme sur $\mathcal{T}(\Sigma_{pub},\mathcal{W})$. Étant donné une *frame* Φ et une recette R telle que $var(R) \subset dom(\Phi)$ (le domaine de Φ), alors $t = \Phi(R)$ est le terme calculé à partir de Φ en appliquant R. Soit le calcul échoue ($t \not\Downarrow$), soit le calcul aboutit à un message u ($t \Downarrow u$). Cela permet donc à l'attaquant de tester des égalités entre plusieurs termes calculés ou bien de tester si un calcul échoue ou non.

Exemple 4 *Considérons la* frame $\Phi = \{w_1 \mapsto k, w_2 \mapsto enc(m,k)\}$ *pour un message* m *quelconque et* $k \in \mathcal{N}$. *On suppose ici que tous les symboles de* Σ *sont publiques. Alors, de cette connaissance, l'attaquant peut déduire* m *à l'aide de la recette* $R_m = dec(w_2,w_1)$ *(car* $\Phi(R_m) \Downarrow m$*). De même, l'attaquant peut observer l'échec d'un calcul avec la recette* $R_{fail} = dec(w_1,w_2)$ *(car* $\Phi(R_{fail}) \not\Downarrow$*).*

1.3 Algèbre de processus

On souhaite à présent modéliser un protocole quelconque. Prenons par exemple le protocole de Feldhofer tel qu'il est présenté dans la thèse de Lucca Hirschi [5]. Deux agents I (l'initiateur) et R (le répondeur) sont impliqués et, à l'issue de son exécution, ce protocole assure l'authentification mutuelle des deux agents (chacun des deux agents connaît l'identité de l'autre agent). Il peut être schématisé ainsi (on note $\{m\}_k$ le chiffrement du message m à l'aide de la clé k) :

1. $I \to R : n_I$
2. $R \to I : \{n_I, n_R\}_k$
3. $I \to R : \{n_R, n_I\}_k$

n_I et n_R sont des nonces frais qui identifient respectivement l'agent I et l'agent R. Dans ce protocole, I envoie à R son identité n_I. R reçoit n_I, l'apparie avec sa propre identité n_R, chiffre le tout à l'aide de la clé k et envoie le message obtenu à I. I déchiffre le message qu'il reçoit, vérifie qu'il est bien formé (en particulier, que son identité correspond bien à la première partie de la paire envoyée) et envoie le même

P, Q	$:=$	0	nul
	$\mid$	$\text{in}(c,x).P$	*input*
	$\mid$	$\text{out}(c,u).P$	*output*
	$\mid$	$P \mid Q$	composition parallèle
	$\mid$	$new\ \overline{n}.P$	restriction
	$\mid$	$!P$	réplication
	$\mid$	$\text{let } \overline{x} = \overline{t} \text{ in } P \text{ else } Q$	évaluation

où c est un nom de canal ($c \in \mathcal{C}$), x est un nom de variable ($x \in \mathcal{X}$), u est un terme sans destructeur $u \in \mathcal{T}\{\Sigma_c, \mathcal{N} \cup \mathcal{X}\}$, $\overline{n}$ est une séquence de noms ($\overline{n} \in \mathcal{N}^*$) et $\overline{x}$ et $\overline{t}$ sont deux séquences de même longueur respectivement de variables ($\overline{x} \in \mathcal{X}^*$) et de termes ($\overline{t} \in \mathcal{T}\{\Sigma, \mathcal{N} \cup \mathcal{X}\}^*$)

FIGURE 1 – Syntaxe d'un processus

message à R en inversant les rôles de n_I et n_R (cela évite un type d'attaque particulier). Enfin, R vérifie que le message qu'il reçoit est celui attendu.

Afin de définir rigoureusement un tel protocole, il faut introduire une syntaxe et une sémantique permettant de modéliser des protocoles cryptographiques. C'est ce qui est fait par la suite.

1.3.1 Syntaxe

On considère un ensemble infini de noms de canal $\mathcal{C}$ tel que $\mathcal{C} \cap \mathcal{N} = \emptyset$. Ces noms de canal sont supposés publiques, c'est-à-dire que l'attaquant y a pleinement accès. Ils font donc partie du réseau sur lequel l'attaquant a de l'influence. C'est sur ces canaux que les agents vont pouvoir communiquer. Un protocole est modélisé en utilisant la syntaxe définie à la figure 1.

Le processus 0 est le processus nul, pendant lequel rien ne s'exécute. Un processus du type $\text{in}(c,x).P$ attend un message u sur le canal c, puis se comporte comme le processus $P\{x \mapsto u\}$ (où u se substitue à x dans le processus P). Un processus $\text{out}(c,t).P$ peut diffuser le message u sur le canal c à condition que $t \Downarrow u$, puis se comporte comme P. La composition parallèle $P|Q$ de processus se comporte comme si P et Q s'exécutaient alternativement (par exemple, P s'exécute en entier, puis Q, ou alors on exécute une action de P, puis une de Q et ainsi de suite). Le processus $new\ \overline{n}.P$ déclare des nonces frais $\overline{n}$ que P va pouvoir utiliser. La réplication $!P$ se comporte comme une composition parallèle infinie $P \mid (P \mid (P \mid ...))$. Enfin, l'évaluation $\text{let } \overline{x} = \overline{t} \text{ in } P \text{ else } Q$ permet d'écrire à la fois un calcul et une conditionnelle. On essaie d'exécuter les différents termes de $\overline{t}$, et s'il existe $\overline{u}$ tel que $\overline{t} \Downarrow \overline{u}$ alors le processus $P\{\overline{x} \mapsto \overline{u}\}$ (où chaque message u_i de $\overline{u}$ se substitue à chaque variable x_i de $\overline{x}$ dans le processus P) s'exécute. Dans le cas contraire (si $\overline{t} \not\Downarrow$), alors le processus Q s'exécute (lorsque Q est le processus nul, on peut se permettre d'omettre else 0 dans l'écriture d'un processus).

Exemple 5 *Reprenons le protocole de Feldhofer défini plus haut. Celui-ci peut être modélisé par le parallélisation de deux processus (les couleurs permettront d'identifier les processus dans l'exemple suivant) :*

$$P_{FH} = new\ k.(new n_I.P_I \mid new n_R.P_R).$$

avec P_I et P_R définis ainsi :

In	$(\{\text{in}(c,x).P\} \cup \mathcal{P}; \Phi) \xrightarrow{in(c,R)} (\{P\{x \mapsto u\}\} \cup \mathcal{P}; \Phi)$ où R est une recette telle que $R(\Phi) \Downarrow u$.
Out	$(\{\text{out}(c,t).P\} \cup \mathcal{P}; \Phi) \xrightarrow{out(c,w)} (\{P\} \cup \mathcal{P}; \Phi \cup \{w \mapsto u\})$ avec $w \in \mathcal{W}$ et $t \Downarrow u$.
Par	$(\{P \mid Q\} \cup \mathcal{P}; \Phi) \xrightarrow{\tau_{par}} (\{P\} \cup (\mathcal{P} \cup \{Q\}); \Phi)$.
New	$(\{new\ \overline{n}.P\} \cup \mathcal{P}; \Phi) \xrightarrow{\tau_{new}} (\{P\} \cup (\mathcal{P}); \Phi)$.
Repl	$(\{!P\} \cup \mathcal{P}; \Phi) \xrightarrow{\tau_{repl}} (\{P, !P\} \cup (\mathcal{P}); \Phi)$.
Let-In	$(\{\text{let } \overline{x} = \overline{t} \text{ in } P \text{ else } Q\} \cup \mathcal{P}; \Phi) \xrightarrow{\tau_{then}} (\{P\{\overline{x} \mapsto \overline{u}\}\} \cup \mathcal{P}; \Phi)$ s'il existe $\overline{u}$ vérifiant $\overline{t} \Downarrow \overline{u}$.
Let-Fail	$(\{\text{let } \overline{x} = \overline{t} \text{ in } P \text{ else } Q\} \cup \mathcal{P}; \Phi) \xrightarrow{\tau_{else}} (\{Q\} \cup \mathcal{P}; \Phi)$ si $\overline{t} \not\Downarrow$.

FIGURE 2 – Sémantique d'un processus

$$
\begin{aligned}
P_I \quad = \quad & out(c, n_I). \\
& in(c, x_I). \\
& let\ x_2,\ x_3\ =\ eq(\ n_I,\ \pi_1(dec(x_1, k))\ ),\ \pi_2(dec(x_1, k))\ in \\
& out(c, enc((x_3, n_I), k)).
\end{aligned}
$$

$$
\begin{aligned}
P_R \quad = \quad & in(c, y_1). \\
& out(c, enc((y_1, n_R), k)). \\
& in(c, y_2). \\
& let\ y_3\ =\ eq(y_2, enc((n_R, y_1), k))\ in\ 0.
\end{aligned}
$$

1.3.2 Sémantique

Il s'agit à présent de définir une sémantique associée à ces différentes actions. Pour cela, il nous faut définir la notion de configuration. Une configuration est une paire $(\mathcal{P}, \Phi)$ où $\mathcal{P}$ est un ensemble de processus à exécuter en parallèle, tandis que Φ est une *frame* qui désigne la connaissance actuelle de l'attaquant. On peut alors voir un processus comme une configuration où la connaissance de l'attaquant est nulle. On peut donc assimiler la configuration $(\{\mathcal{P}_{FH}\}, \emptyset)$ au protocole de Feldhofer. La figure 2 présente une sémantique de processus permettant de décrire comment évolue une configuration en fonction de l'avancement d'un processus.

La notation $\xrightarrow{\alpha}$ désigne la transition entre deux configurations via l'action α. On désigne par trace (souvent notée tr) une séquence d'actions $\alpha_1 ... \alpha_n$. On peut noter que les actions annotées par τ (c'est-à-dire $\tau_{par}, \tau_{new}, \tau_{repl}, \tau_{then}$ et τ_{else}) ne sont pas des actions observables par l'attaquant. Cela représente l'évolution interne des protocoles considérés. On note $obs(tr)$ la sous-séquence obtenue à partir de tr en retirant toutes les actions inobservables.

Exemple 6 *Reprenons l'exemple 5. Une exécution honnête du protocole de Feldhofer pouurait donner :* $(\{P_{FH}\}; \emptyset) \xrightarrow{tr} (\{0\}; \Phi)$ *avec* tr *et* Φ *définis ainsi (*k',n'_I *et* n'_R *étant des noms frais de* $\mathcal{N}$*) :*

$\text{tr} = \tau_{new}.\tau_{par}.\tau_{new}.\tau_{new}.\ out(c, w_1).in(c, w_1).out(c, w_2).in(c, w_2).\tau_{then}.out(c, w_3).in(c, w_3).\tau_{then}$

$\Phi = \{w_1 \mapsto n'_I, w_2 \mapsto enc((n'_I, n'_R), k'), w_3 \mapsto enc((n'_R, n'_I), k')\}$

On a également $obs(tr) = out(c, w_1).in(c, w_1).out(c, w_2).in(c, w_2).out(c, w_3).in(c, w_3)$

2 Objet d'étude : propriétés de sécurité étudiées

Le formalisme nécessaire à l'étude des protocoles cryptographiques a donc été introduit. On souhaite à présent s'en servir pour vérifier qu'un protocole donné assure certaines propriétés.

2.1 Une première propriété de sécurité

On définit ici la propriété de non-traçabilité d'un protocole. On introduit préalablement quelques notions d'équivalence qui permettront de définir convenablement cette propriété.

2.1.1 Équivalence statique et Équivalence de trace

Comme évoqué plus tôt, l'attaquant intercepte tous les messages circulant sur le réseau, et peut effectuer à sa guise toute une batterie de tests, en appliquant certaines fonctions publiques et en testant l'égalité de termes à sa disposition. Ces tests peuvent lui permettre de distinguer différentes *frame*s. On peut dire que deux *frame*s Φ et Ψ sont indistinguables (modulo une théorie équationnelle E) lorsqu'elles vérifient :

- pour toutes recettes R telle que $R(\Phi) \Downarrow u$ pour un message donné u, il existe un message u' tel que $R(\Psi) \Downarrow u'$ (et réciproquement, en inversant les rôles de Φ et Ψ).
- s'il existe deux recettes R_1 et R_2 telles que $R_1(\Phi) \Downarrow u_1$ et $R_2(\Phi) \Downarrow u_2$ pour deux messages u_1 et u_2 vérifiant $u_1 =_E u_2$, alors il existe v_1 et v_2 tels que $R_1(\Psi) \Downarrow v_1$, $R_2(\Psi) \Downarrow v_2$ et $v_1 =_E v_2$ (et réciproquement, en inversant les rôles de Φ et Ψ).

Cela signifie que l'attaquant peut distinguer deux *frame*s s'il peut effectuer un calcul qui aboutit à un message sur une *frame*, alors que ce même calcul échoue sur l'autre. Lorsque deux *frame*s Φ et Ψ sont indistinguables, on dit qu'elles sont en équivalence statique, notée $\Phi \sim \Psi$.

Exemple 7 *On reprend la théorie équationnelle de l'exemple 2, et le système de réécriture de l'exemple 3. Alors les deux* frames $\Phi = \{w_1 \mapsto k\}$ *et* $\Psi = \{w_1 \mapsto k'\}$ *sont indistinguables. En revanche, les deux* frames $\Phi' = \Phi \cup \{w_2 \mapsto enc(n,k)\}$ *et* $\Psi' = \Psi \cup \{w_2 \mapsto enc(n,k)\}$ *ne le sont pas. En effet, la recette* $R = dec(w_2, w_1)$ *aboutit à un message lorsqu'elle est appliquée à* Φ' *mais échoue lorsqu'elle est appliquée à* Ψ'.

L'équivalence statique est une relation entre deux *frame*s. On peut également définir une équivalence entre deux configurations. Ainsi, on dit que deux configurations K et K' sont en équivalence de trace (notée $K \approx K'$) si, quelque soit la *frame* Φ obtenue en appliquant une trace tr_1 à la configuration K, il existe une trace tr_2 ayant les mêmes actions observables que tr_1 telle que la *frame* Ψ obtenue en appliquant tr_2 à K' vérifie $\Phi \sim \Psi$, et réciproquement, en inversant les rôles de Φ et Ψ. Cela signifie que, peu importe les actions de l'attaquant sur les configurations en question, il ne peut distinguer les *frame*s qu'il obtient. Cette définition s'étend naturellement aux processus étant donné que, comme évoqué précédemment, un processus peut être vu comme une configuration. Il est important de remarquer que cette définition s'intéresse uniquement aux actions observables par l'attaquant (*in* et *out*) et aux *frame*s résultantes.

2.1.2 Non-traçabilité

On peut, à présent, définir convenablement la propriété de non-traçabilité. Informellement, cette propriété vise à assurer qu'un utilisateur puisse faire usage d'un service à plusieurs reprises sans que

ceux-ci puissent être liés entre eux. C'est-à-dire qu'il n'y a pas moyen de relier ces multiples usages à un même utilisateur. Cela est modélisé par le fait que les deux situations suivantes sont indistinguables : d'une part, une situation dans laquelle chaque utilisateur peut faire usage du service autant de fois qu'il le désire et d'autre part une situation au cours de laquelle chaque utilisateur ne peut utiliser ce même service qu'une seule fois. Cette définition peut se traduire formellement à l'aide de la notion d'équivalence de trace pour un processus à deux parties, c'est-à-dire de la forme $P = new\ \overline{n}.(!P_I \mid !P_R)$ (le protocole de Feldhofer en est un). Un tel processus vérifie la propriété de non-traçabilité si l'équivalence suivante est vérifiée :

$!\text{new}\ \overline{n};\ (!P_I \mid !P_R)$	$\approx$	$!\text{new}\ \overline{n};\ (P_I \mid P_R)$
Situation réelle : potentiellement plusieurs sessions par individu.		Situation idéalisée : uniquement une session par individu.

Une telle propriété d'équivalence peut se révéler ardue à vérifier à la main pour un protocole donné. C'est pourquoi on souhaite pouvoir utiliser un outil de vérification automatique. L'un des outils adaptés au formalisme du π-calcul appliqué est Proverif [2].

2.2 L'outil Proverif

Proverif est un outil de vérification automatique de protocoles cryptographiques. Toutes les notions définies à la partie 1 peuvent être inclues dans la définition d'un processus en Proverif. En effet, il est possible de définir des symboles de fonction, de générer de nouveaux noms de canal, ou encore des nonces frais. Par exemple, on peut déclarer une théorie équationnelle à l'aide du mot clé equation. De même, un système de réécriture peut être déclaré à l'aide du mot clé reduc. Un exemple de protocole codé en Proverif se trouve en annexe A (il s'agit du protocole de Feldhofer). Un fois le protocole spécifié, on peut effectuer un certain nombre de requêtes à Proverif. Par exemple, la requête query attacker(new nI) demande à Proverif si l'attaquant a accès au nonce nI qui sera déclaré plus tard dans le processus (dans le cas du protocole de Feldhofer, c'est évidemment le cas, puisqu'il est émis sur le canal publique ci).

Proverif fonctionne en utilisant la mise sous forme de clauses de Horn. Ces clauses sont de la forme $(h_1 \wedge h_2 \wedge ... \wedge h_2 \Rightarrow c\ )$ et traduisent le fait que sous les hypothèses h_1, h_2, ... h_n on peut déduire c. Ainsi, lorsque l'on déclare une fonction publique f, d'arité 2 par exemple, cela revient à déclarer la clause $attacker(a) \wedge attacker(b) \Rightarrow attacker(f(a,b))$ (la construction $attacker(m)$ signifie que l'attaquant connaît m). Le même processus peut être adopté pour les définitions d'équation ou de système de réécriture. Une fois que le protocole a été entièrement abstrait en un ensemble de clauses de Horn, il faut travailler sur celles-ci afin de prouver la propriété demandée. Il s'agit d'une représentation interne à Proverif que l'on ne manipule pas directement. Il faut tout de même noter que Proverif n'a pas de garantit de terminaison ni de complétude car les problèmes liés à la connaissance de l'attaquant sont, en général, indécidables.

Cependant, Proverif est assez efficace en pratique lorsqu'il s'agit de prouver une propriété d'atteignabilité (en terme de connaissance de l'attaquant). Toutefois, la manière dont Proverif gère la notion d'équivalence est moins performante. La seule notion d'équivalence que Proverif puisse manipuler est la diff-équivalence. Celle-ci ne s'applique pas à deux processus mais plutôt à un bi-processus. Celui-ci se comporte comme un processus classique à l'exception des termes présents dans les *output*s, car ceux-ci sont de la forme choice[u1,u2]. La sémantique associée est celle de la figure 3.

On peut remarquer que ces règles ne s'exécutent que si les deux côtés du choice produisent le même résultat. En effet, il faut ou bien que les deux termes s'exécutent sans échouer, ou bien que

Let-In $(\{\text{let } choice[\overline{x_1}, \overline{x_2}] = choice[\overline{t_1}, \overline{t_2}] \text{ in } P \text{ else } Q\} \cup \mathcal{P}; \Phi) \xrightarrow{\tau_{then}}$
$(\{P\{choice[\overline{x_1}, \overline{x_2}] \mapsto choice[\overline{u_1}, \overline{u_2}]\}\} \cup \mathcal{P}; \Phi)$ si $\overline{t_1} \Downarrow \overline{u_1}$ et $\overline{t_2} \Downarrow \overline{u_2}$.

Let-Fail $(\{\text{let } choice[\overline{x_1}, \overline{x_2}] = choice[\overline{t_1}, \overline{t_2}] \text{ in } P \text{ else } Q\} \cup \mathcal{P}; \Phi) \xrightarrow{\tau_{else}} (\{Q\} \cup \mathcal{P}; \Phi)$ si $\overline{t_1} \not\Downarrow$ et $\overline{t_2} \not\Downarrow$.

FIGURE 3 – Sémantique des règles conditionnelles d'un bi-processus

l'exécution échoue des deux côtés. Pour qu'il y ait diff-équivalence, il faut que le bi-processus puisse s'exécuter (c'est-à-dire que les règles conditionnelles ne posent pas problème), et que les deux *frame*s obtenues en ne considérant que la partie gauche ou que la partie droite des choice soient en équivalence statique. Il est également possible de transformer deux processus distincts en un bi-processus pour tester s'ils sont en diff-équivalence. Cependant, cette notion d'équivalence est fondamentalement différente de l'équivalence de trace qui intervient dans la définition de non-traçabilité. En effet, la diff-équivalence prend en compte, dans sa définition, des actions inobservables telles que τ_{then} ou τ_{else}. Il s'agit donc d'une propriété plus beaucoup forte que l'équivalence de trace, car on ne requiert pas seulement que les *output*s des processus soient indistinguables, il faut en outre que la structure interne des processus soit analogue. C'est pourquoi il n'est pas possible d'utiliser directement Proverif pour prouver qu'un processus donné assure la propriété de non-traçabilité.

2.3 Une propriété plus faible de sécurité

Il faut donc utiliser Proverif pour prouver des conditions plus faibles qui pourront, sous certaines conditions, impliquer la propriété de non-traçabilité. C'est, en partie, l'objet de la thèse de Lucca Hirschi [5]. Il a établi que pour une large classe de protocoles à deux parties (le protocole de Feldhofer en est un), si un protocole assure à la fois *Frame Opacity* et *Well-Authentication* alors il n'est pas traçable. On ne détaille pas ici la propriété de *Well-Authentication* car ce n'était pas l'objet de ce stage. On peut tout de même dire qu'il s'agit d'une propriété d'atteignabilité que Proverif gère plutôt efficacement.

On s'intéresse donc à la propriété de *Frame Opacity*. Cette propriété vise à assurer que, à tout instant, les connaissances de l'attaquant ne dépendent pas des identités spécifiques des agents qui interviennent dans le protocole. Cela se traduit par le fait que toute *frame* atteignable est indistinguable d'une *frame* idéale qui ne dépend que des informations déjà observées, et pas d'agent spécifique.

Cette *frame* idéale doit respectée un certain nombre de contraintes. Lorsqu'on idéalise une *frame*, on ne fait que changer les *ouput*s, sans en rajouter, ni en enlever. Ainsi, le domaine de la *frame* idéale est le même que celui de la *frame* réelle. En outre, on doit s'assurer que tous les noms d'agent ont disparu. Il est à noter que l'on peut définir de différentes façons la *frame* idéale tant que le choix effectué respecte les conditions précédentes. L'idéalisation la plus simple consiste à idéaliser tous les termes par des nonces frais. Cependant, il apparaît que si on idéalise un tuple par un nonce, alors on les *frame*s seront distinguables car l'attaquant pourra tester l'application de fonctions de projection qui réussiront du côté réel mais échoueront du côté de l'idéalisation. Il faut trouver une idéalisation licite qui permette de conclure à l'équivalence statique.

Exemple 8 *On se place dans le cadre de la théorie équationnelle de l'exemple 2, et du système de réécriture de l'exemple 3.*

Reprenons alors l'exemple 6, avec la frame *:*

$\Phi = \{w_1 \mapsto n'_I, w_2 \mapsto enc((n'_I, n'_R), k'), w_3 \mapsto enc((n'_R, n'_I), k')\}$.

Celle-ci peut être idéalisée par la frame $\Phi_{ideal} = \{w_1 \mapsto n_1, w_2 \mapsto n_2, w_3 \mapsto n_3\}$ *(avec* n_1,n_2 *et*

n_3 nonces frais). On peut remarquer que les noms d'agent ne sont plus présents dans la frame *idéalisée. Elle permet bien de conclure : on a $\Phi \sim \Phi_{ideal}$ (on peut s'en convaincre en remarquant que l'attaquant n'a pas de clé à sa disposition, le déchiffrement de w_2 et w_3 échouera donc dans le cas idéal comme dans le cas non idéal). Considérons à présent la* frame $\Psi = \{w_1 \mapsto n'_I, w_2 \mapsto (n'_I, n'_R)\}$ *à idéaliser (avec n'_I et n'_R des noms d'agent). Dans ce cas, la* frame $\Psi^1_{ideal} = \{w_1 \mapsto n_1, w_2 \mapsto n_2\}$ *est bien une idéalisation de Ψ (avec n_1 et n_2 nonces frais). Toutefois, l'équivalence $\Psi \sim \Psi^1_{ideal}$ ne tient pas car $\pi_1(w_2) \Downarrow n'_I$ pour Ψ alors que $\pi_1(w_2) \not\Downarrow$ pour Ψ^1_{ideal}. Cependant, la* frame $\Psi^2_{ideal} = \{w_1 \mapsto n_1, w_2 \mapsto (n_1, n_2)\}$ *est encore une idéalisation valide de Ψ et, cette fois, on a bien $\Psi \sim \Psi^2_{ideal}$.*

On dit qu'un protocole $\mathcal{P}$ assure *Frame Opacity* si, pour toute exécution $\{\mathcal{P}, \emptyset\} \xrightarrow{tr} \{\mathcal{Q}, \Phi\}$, on a : $\Phi_{ideal}(tr) \sim \Phi$. Or, il a été évoqué que l'on peut définir plusieurs idéalisations différentes. Il est donc important de remarquer que pour avoir *Frame Opacity*, il faut qu'il existe au moins une idéalisation permettant d'établir l'équivalence présentée plus haut. En pratique, pour établir *Frame Opacity*, il faut trouver l'idéalisation qui convient.

Ainsi, l'équivalence de trace qui intervenait dans la définition de non-traçabilité a été remplacé par la notion, plus faible, d'équivalence statique. Cela est préférable car, à présent, on ne travaille plus qu'avec un seul processus. Il semble donc faisable de le modifier en un bi-processus, pour pouvoir utiliser la diff-équivalence de Proverif.

3 Contribution

On présente, dans un premier temps, le codage Proverif sur lequel j'ai travaillé durant ce stage. Dans un second temps, on s'intéressera à l'implémentation effective de celui-ci.

3.1 Codage proposé

On se donne un protocole cryptographique P respectant la syntaxe de protocole définie plus tôt. Celui-ci peut être codé par un processus Proverif P. On cherche alors à modifier P pour pouvoir s'assurer que P assure *Frame Opacity*. Pour cela, on mettra en place le processus Proverif suivant :

```
P_FO = ( P' | attacker | phase 1; P_Check).
```

P' correspond au processus P dans lequel tous les out(c,u) ont été remplacés par out(c_attack,u); out(c_public,enc((u,ideal(u)),K_priv)) avec ideal(u) correspondant à l'idéalisation de u. K_priv est une clé privée (à laquelle l'attaquant n'a pas accès). De même, le canal c_attack est privé. En revanche, le canal c_public est publique, même si l'attaquant ne peut récupérer directement les messages u et ideal(u) car il n'est pas en possession de la clé K_priv. Il est à noter que la fonction enc utilisée ici doit être définie plus tôt si elle n'existe pas déjà (tout comme la fonction de déchiffrement associée dec).

Le processus attacker simule un attaquant, selon la définition de l'approche symbolique. Celui-ci travaille sur le canal privé c_attack et peut envoyer sur celui-ci tous les messages qu'il a pu calculer à partir de ceux qu'il a déjà récupérés sur ce même canal (il ne travaille qu'avec des fonctions publiques).

Enfin, le processus P_Check consiste en la récupération de tous les messages enc((u,ideal(u)),K_priv) circulant sur le canal c_public, à leur déchiffrement à l'aide de la clé K_priv puis à leur envoie sur le canal c_public sous la forme choice[u,ideal(u)].

La construction phase 1 est spécifique à Proverif. Par défaut, tous les processus sont en phase 0 (c'est en particulier le cas pour P' et attacker). L'intérêt vient du fait que les processus en phase 1 ne peuvent pas être suivis de processus en phase 0. Dans le cas qui nous occupe, on exécute parallèlement les processus P' et attacker et, lorsque l'on passe au processus phase 1; P_Check, on ne peut plus revenir en arrière. Ainsi l'exécution des processus P' et attacker doit être finie avant de passer à l'exécution du processus P_Check.

L'idée de ce codage est que l'on simule l'exécution du processus P, dans lequel l'attaquant intervient via le canal c_attack. Celui-ci a exactement les mêmes capacités que l'attaquant Proverif. Une fois cette simulation achevée, on cherche à établir l'équivalence entre la *frame* réelle (côté gauche du choice) et la *frame* idéale (côté droit du choice).

Ce que fait le processus attacker, Proverif est en capacité de le faire tout seul (en travaillant sur un canal publique). Pourtant, il est essentiel de travailler sur un canal privé. En effet, supposons que l'on travaille sur un canal publique, et donc que Proverif ait accès directement aux messages échangés. Alors, ceux-ci sont disponibles pour une comparaison des deux côtés des choice qui apparaîtront. Ainsi, on aura égalité entre la partie gauche des choice et ce qui a déjà été observé, tandis qu'il y aura inégalité avec la partie droite. C'est pourquoi, un tel codage ne pourrait jamais conclure. L'attaquant doit donc être simulé en utilisant un canal privé.

La notion de diff-équivalence que Proverif utilise ne pose pas de problème ici puisque l'on ne travaille qu'avec un processus dans lequel on n'a modifié que les messages émis. Ainsi, si Proverif établit la diff-équivalence pour le processus P_FO, alors le protocole cryptographique P assure *Frame Opacity*.

Un exemple d'un tel codage se trouve en annexe B. Il s'agit de la modification du protocole de Feldhofer présent en annexe A.

3.2 Implémentation

Afin d'automatiser le codage présenté ci-dessus, un outil a été implémenté en OCaml. Ce langage a été choisi car il m'est familier. De plus, l'implémentation de Lucca utilisait déjà ce langage, ainsi il m'était possible de m'inspirer de certaines parties pour ma propre implémentation. Cet outil s'appuie sur un *parser* et un *lexer* permettant de récupérer les informations relatives à un protocole donné. Leur traitement permet alors de générer un nouveau fichier sur lequel est alors lancé Proverif. Si Proverif conclut à la diff-équivalence, cela signifie que le protocole initial vérifie la propriété de *Frame Opacity*. La grammaire utilisée pour écrire le *parser* est celle décrite au sein d'un manuel consacré à Proverif [3]. Il est à noter que, bien que cette grammaire ait été respectée, un certain nombre de constructions Proverif n'ont pas été prises en compte, car celles-ci ne rentraient pas dans le cadre d'étude de ce stage. Par exemple, la construction query, qui correspond à une requête Proverif concernant une propriété d'atteignabilité, est incompatible avec la construction choice utilisée pour vérifier la diff-équivalence.

Pour réaliser le codage initial, Lucca Hirschi a déjà implémenté un *parser* et un *lexer* permettant de générer un protocole Proverif pour vérifier *Frame Opacity* et *Well-Authentication*. Il aurait été préférable de le réutiliser pour faciliter l'intégration de ce nouvel outil au sein du travail déjà en place. Cependant, celui-ci était complexe et le temps d'appropriation de ce *parser* aurait été sans doute supérieur à l'écriture d'un nouveau. De plus, comme le temps de transformation d'un protocole P en son protocole associé P_FO est infime (étant donné la taille, en nombre de lignes, des protocoles considérés) il n'est pas réellement gênant d'utiliser deux implémentations différentes pour modifier un même protocole. L'implémentation finale comprend plus 1500 lignes de code (commentaires compris), réparties sur quatre fichiers. L'implémentation et les fichiers sur lesquels on a travaillé sont disponible à l'adresse https://drive.google.com/open?id=12WqILWy5C7Se6FkN_PDj7BAT3G-RSSHI.

L'implémentation se décompose en deux parties. Tout d'abord, il a fallu récupérer les informations permettant de simuler l'attaquant. En effet, il faut avoir à sa disposition tous les noms de fonction publique, ainsi que leur arité pour pouvoir permettre à l'attaquant d'effectuer tous les calculs qu'il a le droit de faire. En outre, l'ensemble des constantes publiques déclarées doivent être accessibles à l'attaquant. Enfin, la déclaration de tous les canaux doit être prise en compte (c'est au moment de la déclaration que l'on précise si un canal est publique ou privé), car ce qui circule sur un canal privé ne doit jamais être diffusé sur le canal publique sous la forme enc((u,ideal(u)),K_priv), car le message sera alors déchiffré (au sein du processus P_Check). En outre, il est à noter qu'il faut également rajouter la possibilité à l'attaquant de créer des tuples et de calculer des projections. Cela est dû au fait que ces fonctions sont toujours disponibles (et publiques) en Proverif, sans qu'il y ait besoin de les déclarer. Il faut donc repérer l'arité de tous les tuples apparaissant dans le protocole considéré afin de générer les fonctions d'appariement et de projection adéquates.

Ainsi, la simulation de l'attaquant requiert surtout un travail sur les déclarations précédant les processus eux-mêmes. *A contrario*, pour générer une idéalisation convenable, il faut se pencher principalement sur le contenu des processus. On doit repérer les out afin de les modifier correctement. Il faut donc vérifier si des tuples y apparaissent, pour les idéaliser convenablement. En outre, on n'idéalise pas de la même manière un nonce correspondant à un nom d'agent (qui doit être remplacé par un nonce frais préalablement déclaré) et un nonce frais quelconque (qui doit être laissé tel quel). De plus, les constantes publiques doivent également être conservées.

Toutes les informations nécessaires à la simulation de l'attaquant et à une idéalisation convenable sont récupérées lorsque le *parser* traite le fichier d'entrée. Elles sont stockées par effet de bords dans différentes listes prévues à cet effet. Cela permet de générer ensuite les processus Proverif adaptés.

Le codage Proverif de l'annexe B correspond au résultat obtenu si on lance l'outil implémenté sur le fichier de l'annexe A (à l'exception des commentaires, qui ont été rajoutés pour que le fichier soit plus lisible). Cela correspond bien au codage proposé.

4 Validation

Avant d'évaluer l'efficacité du codage proposé, il a fallu s'assurer de la correction de l'implémentation mise en place. Il faut en premier lieu préciser ce que l'outil est censé faire. Il prend en entrée un codage Proverif, ne comportant aucune des constructions non autorisées (celles-ci sont listées dans un *Readme*). Un nouveau fichier est alors généré. Il correspond au codage permettant de vérifier que le protocole initial, implémenté en Proverif, vérifie la propriété de *Frame Opacity*. Il faut donc vérifier deux choses : l'outil ne renvoie pas d'erreur lorsque le fichier pris en entrée est correct, et le fichier généré à partir de celui-ci correspond au codage souhaité.

Les erreurs que peut générer le lancement de l'outil sur un fichier ne peuvent provenir que du *lexer* ou du *parser*. Chaque construction Proverif a été vérifiée individuellement afin de s'assurer qu'elle était correctement traitée par l'outil. De plus, il a été vérifié sur des protocoles implémentés en Proverif qu'aucune erreur ne survenait. Ces tests ont été réalisés sur les fichiers utilisés par Lucca Hirschi pour évaluer l'efficacité de son codage. Une garantie supplémentaire vient du fait que la grammaire utilisée par le *parser* respecte la grammaire de Proverif.

Vérifier que le fichier généré correspond au codage souhaité est plus ardu. En effet, lorsque l'on implémente un protocole en Proverif, il n'y a pas de moyen direct pour tester si celui-ci est correct. Cela vient du fait qu'il n'y a pas de résultat généré par cette implémentation. On peut tout de même vérifier, par exemple, que l'ensemble du protocole est accessible (ce qui peut montrer qu'il n'y a pas de

Protocole	Nouveau codage	Codage de Lucca
Feldhofer	<1s	<1s
Hash-Lock	<1s	<1s
LAK	<1s	<1s
BAC	<1s	6-7s
BAC+AA+PA	<1s	112-115s
BAC+PA+AA	<1s	106-111s
PACE	30-33s	65-67s

FIGURE 4 – Tableau comparatif des temps d'exécution de Proverif (en seconde), moyennés sur 30 tentatives en tenant compte de l'écart-type, pour différents protocoles

conditionnelle toujours vraie ou toujours fausse sans raison). Ce type de tests peut se réaliser en utilisant la construction event(reach) qui demande à Proverif si l'événement reach est atteignable. Cependant, cela n'assure pas de garantie absolue. Il faut vérifier à la main que le codage correspond bien à ce que l'on veut. Cependant, la justesse du codage a été vérifiée de nombreuses fois, sa correction est donc très probable.

4.1 Résultats positifs

Il faut à présent évaluer la performance du codage mis en place. On présente dans la figure 4 un tableau récapitulatif des temps d'exécution de Proverif sur mon codage ainsi que sur celui de Lucca Hirschi (l'outil de Lucca peut se trouver à l'adresse https://github.com/LCBH/UKano.git) pour différents protocoles. Tous ces tests ont été effectués sur une même machine pour qu'ils soient bien comparables. Les temps sont présentés avec une précision assez faible car il n'est pas nécessaire (et particulièrement pertinent) d'être plus précis. En effet, on s'intéresse à une tendance générale plutôt qu'à une étude comparative au centième de seconde près. Au vu des résultats, on peut constater que ce nouveau codage est globalement plus efficace, notamment pour les protocoles de type BAC et le protocole PACE. Il faut tout de même remarquer que les temps présentés ici sont tous raisonnables au sens où ces calculs ne doivent être effectués qu'une seule fois pour s'assurer qu'un protocole vérifie une propriété donnée. Il n'est donc pas problématique d'attendre une heure, par exemple, pour avoir un résultat. On peut tout de même supposer que, pour des protocoles qui nécessiteraient plus de calculs, le nouveau codage terminerait plus rapidement.

Ces résultats doivent également être relativisés du fait de la sensibilité de Proverif à l'emplacement des new. En effet, lorsque l'on déclare un nonce à l'aide de cette construction, celui-ci est supposé être frais. Cela n'est, en général, pas le cas dans Proverif. Plus le new est placé vers le bas, plus celui-ci dépend de ce qui a été déclaré précédemment, et donc le nonce généré aura plus de chances d'être réellement frais. Cependant, cela entraine une augmentation du nombre de calculs à effectuer car ces déclarations dépendent alors de nombreux paramètres. Ainsi, plus un new est placé vers le bas, plus Proverif aura de chances de conclure, mais Proverif sera également plus long à terminer. Par défaut, dans le nouveau codage, les new sont placés le plus bas possible. Il est possible qu'un jeu sur l'emplacement de ceux-ci permettrait de diminuer le temps d'exécution (cela a été testé succinctement : en général, remonter les new aboutit à une non-terminaison de Proverif). Cela peut engendrer quelques variations dans le temps d'exécution de Proverif. Toutefois, cela ne remet pas en cause le fait que le nouveau

codage soit globalement plus efficace, sur ces exemples, que celui de Lucca.

Ce nouveau codage semble donc concluant puisqu'il permet de terminer plus rapidement que le codage précédent. Cependant, celui-ci a des limites et ne peut pas être utilisé sur tous les protocoles.

4.2 Limites

En fait, ce codage ne teste la propriété de *Frame Opacity* que pour une certaine classe d'idéalisation. Cela vient du fait que la définition précise d'une idéalisation n'est pas respectée. En effet, l'attaquant travaille toujours avec la partie réelle des messages. Ainsi, les variables d'*input* correspondent également à la partie réelle des messages. Cela est nécessaire car les conditionnelles à l'intérieur des processus ne doivent pas être affectées par la partie idéalisée. Cependant, cela pose problème lorsqu'une variable d'*input* apparaît dans un *output* car alors celle-ci sera mal idéalisée (car on ne connaît pas sa forme, une variable d'*input* quelconque pourrait très bien faire référence à un tuple que l'attaquant a envoyé sur le réseau). Le problème a été observé sur le protocole DAAsign. Sur celui-ci, le nouveau codage ne permettait pas d'établir *Frame Opacity*, à la différence du codage de Lucca qui, lui, y parvenait. Il est à noter que ce problème n'intervient pas chez Lucca en raison de la structure de son codage. Ce phénomène est également présent dans le protocole DAAjoin. Il faut bien remarquer que lorsqu'il y a une variable d'*input* dans un *output* (après idéalisation), montrer la diff-équivalence ne montre pas que le protocole assure *Frame Opacity* car la définition n'est pas respectée. Toutefois, le phénomène en question n'intervient dans aucun des protocoles de la figure 4. C'est pourquoi ces résultats sont toujours valides. Le nouveau codage est donc plus efficace, mais il faut vérifier préalablement que l'on peut l'utiliser, ce qui n'est pas toujours le cas, alors que le codage de Lucca peut être utilisé sur tous les protocoles.

D'autre part, il est possible que le codage ne permette pas d'établir *Frame Opacity*. En effet, l'idéalisation mise en place, bien que licite, peut ne pas être suffisante pour conclure. C'est par exemple le cas pour le protocole Irma. Dans ce cas, la diff-équivalence n'est pas établie (pour le nouveau codage comme pour celui de Lucca). C'est pourquoi d'autres idéalisations ont été mises en place. Dans l'outil de Lucca, trois idéalisations sont disponibles : *greedy* (celle utilisée jusqu'ici), *middle* et *full-syntax*. Cela correspond à trois niveaux de précision dans l'idéalisation, *greedy* étant la plus grossière et *full-syntax* étant la plus précise. Plus l'idéalisation est précise, plus on a de chances de pouvoir établir *Frame Opacity*. Cependant, le temps d'exécution de Proverif sera d'autant plus long. Ces idéalisations ont également été mises en place au sein du nouveau codage. Ainsi, seule l'idéalisation *full-syntax* permet d'établir *Frame Opacity* pour le protocole Irma. Cependant, dans le nouveau codage, avec cette idéalisation, le problème décrit précédemment réapparaît (ce qui n'est pas le cas pour une idéalisation *greedy*). Finalement, le nouveau codage ne peut pas non plus établir *Frame Opacity* pour le protocole Irma.

Pour résoudre ces problèmes, on a envisagé plusieurs manières de modifier légèrement le codage. La première solution consistait à tester, dans les fonctions simulant l'attaquant, la distinguabilité entre les parties réelles et idéalisées. Cela permet de se ramener à un problème d'atteignabilité. Même si cela semblait prometteur en apparence (du fait que Proverif est généralement plus à l'aise avec ce genre de propriétés), Proverif n'est pas capable de conclure quoi que ce soit avec ce codage. Une deuxième idée consistait à travailler en permanence avec des paires dans les *input*s et les *output*s. A tout instant, la partie gauche de la paire correspond à la partie réelle, et la partie droite à son idéalisation. Quand une variable d'*input* apparaît dans un *output*, elle peut, à présent, être correctement idéalisée. Ce codage permet d'éviter le problème évoquée deux paragraphes plus haut. On peut, par conséquent, mettre en place les trois idéalisations pour tous les protocoles. Un exemple de ce codage se trouve en annexe C. Il est appliqué au protocole de Feldhofer. Cet autre codage a également été implémenté, en changeant relativement peu de choses par rapport à l'implémentation initiale. Les résultats sont assez prometteurs

pour les protocoles de type BAC (il conclut plus vite que celui de Lucca), mais pour les autres, il semble moins efficace.

Finalement, le codage de base est le plus rapide de tous, mais il ne peut pas être appliqué à tous les protocoles. La variante de ce codage et le codage de Lucca sont effectifs sur tous les protocoles mais sont en général moins efficaces. Il est à noter qu'une fonctionnalité a été rajoutée au codage de base qui vérifie si l'idéalisation choisie est bien licite (on vérifie qu'aucune variable d'*input* n'est présente dans un *output*, après idéalisation). Cela permet de s'assurer que les calculs effectués sont bien utiles.

Conclusion

Le but de ce stage était d'étudier un nouveau codage testant la propriété de *Frame Opacity* pour un protocole cryptographique donné. Afin de tester efficacement ce nouveau codage, nous avons été amenés à automatiser une transformation de protocoles afin d'évaluer plus rapidement les performances de ce codage. Cela nous a conduit à utiliser un *lexer* et un *parser* pour pouvoir transformer correctement les protocoles à notre disposition.

Les performances de ce nouveau codage sont satisfaisantes. En effet, sur une certaine classe de protocoles (ceux sur lesquels l'idéalisation *greedy* est licite et permet de conclure), il semble subsumer le codage de Lucca (en terme de temps d'exécution). Cependant, il possède plusieurs défauts : il ne peut être utilisé sur tous les protocoles, et ne permet pas toujours de conclure.

Ce travail peut être poursuivi de différentes manières. On peut, par exemple, essayer de transformer légèrement ce codage pour qu'il se base sur l'atteignabilité d'un événement. En effet, sur ce genre de propriétés, Proverif est en général plus efficace. Cependant, après quelques tentatives infructueuses, cela semble ardu. Ainsi, ce qui est envisagé pour la suite est assez différent. Il s'agit de modifier superficiellement l'outil Proverif pour autoriser le fait d'écrire des choice dans les *input*s. Cela pourrait permettre de conserver l'efficacité du nouveau codage tout en étendant son champ d'action. Si une telle modification s'avérait efficace, l'outil implémenté durant ce stage pourrait être légèrement modifié afin de mettre en place cette nouvelle approche.

Références

[1] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L.. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on : breaking the SAML-based single sign-on for Google apps. In *Proc. 6th ACM Workshop on Formal Methods in Security Engineering (FMSE'08)*, pages 1–10. ACM, 2008.

[2] Bruno Blanchet. Proverif : Cryptographic protocol verifier in the formal model, 2002. http://proverif.inria.fr.

[3] Blanchet Bruno, Smyth Ben, Cheval Vincent, and Marc Sylvestre. *ProVerif 1.96 :Automatic Cryptographic Protocol Verifier,User Manual and Tutorial*. Inria, 2016.

[4] Rémy Chrétien, Véronique Cortier, and Stéphanie Delaune. From security protocols to pushdown automata. In Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg, editors, *Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP'13) – Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 137–149, Riga, Latvia, July 2013. Springer.

[5] Lucca Hirschi. *Vérification automatique de la protection de la vie privée : entre théorie et pratique*. PhD thesis, Université Paris-Saclay, 2017.

[6] Lucca Hirschi and Stéphanie Delaune. A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols. *Journal of Logical and Algebraic Methods in Programming*, 87 :127–144, 2017.

[7] Cortier Véronique and Kremer Steve. *Formal Models and Techniques for Analyzing Security Protocols : A Tutorial*. Foundations and Trends® in Programming Languages, 2014.

A Protocole de Feldhofer en Proverif

```
(* 'bitstring' est un nom de type générique qui désigne n'importe quel terme *)
(**********)
(* THEORIE *)
(**********)

(* Canaux de communication publiques *)
free ci : channel.
free cr : channel.
(* Constante(s) *)
free ok : bitstring.

(* Fonctions de chiffrement et de déchiffrement *)
fun enc(bitstring, bitstring) : bitstring.
        reduc forall xm:bitstring, xk:bitstring;
                dec(enc(xm, xk), xk) = xm.

(**************)
(* PROTOCOLES *)
(**************)

(* Protocole de Feldhofer *)
let I (k:bitstring) =
        new nI:bitstring;
        (* I -----[n]---> R *)
        out(ci,nI);
        (* I <---[{nI,nR}_k]--- R *)
        in(ci, x:bitstring);
        let (=nI, xnr:bitstring) = dec(x, k) in
        (* I -----[{nR,nI}_k]---> R *)
        out(ci, enc((xnr,nI),k)).

let R (k:bitstring) =
        new nR:bitstring;
        (* R <---[nI]--- I *)
        in(cr, ynI:bitstring);
        (* R -----[{nI,nR}_k]---> I *)
        out(cr, enc((ynI, nR), k));
```

```
        (* R <---[{nR,nI}_k]--- I *)
        in(cr, y:bitstring);
        let (=nR,=ynI) = dec(y,k) in
        out(cr, ok).

(* Système global *)
let FH =
        !new k:bitstring;
        !(I(k) | R(k)).

(* Appel le processus qui doit être exécuté *)
process FH
```

B Protocole de Feldhofer testant *Frame Opacity* en Proverif

```
(* Nouveaux canaux de communications *)
free c_public__ : channel.
free c_attack__ : channel [private].

(* Clé privée *)
free key__ : bitstring [private].

(* Fonctions de chiffrement pour le nouvel attaquant *)
fun atck_enc__(bitstring, bitstring) : bitstring [private].
        reduc forall x : bitstring, y : bitstring;
                atck_dec__(atck_enc__(x, y), y) = x.

(* Ces canaux ne sont plus utilisés *)
free ci : channel.
free cr : channel.

(* Constante(s) *)
free ok : bitstring.

(* Fonctions de chiffrement et de déchiffrement pour le protocole *)
fun enc(bitstring, bitstring) : bitstring.
        reduc forall xm : bitstring, xk : bitstring;
                dec(enc(xm, xk), xk) = xm.

(* Protocole de Feldhofer modifié *)
let I(k:bitstring) =
        new nI : bitstring;
        out(c_attack__, nI);
        out(c_public__, atck_enc__((nI, nI), key__));
        in (c_attack__, x:bitstring);
        let (=nI, xnr:bitstring) = dec(x, k) in
        new hole__0 : bitstring;
        out(c_attack__, enc((xnr, nI), k));
```

```
        out(c_public__, atck_enc__((enc((xnr, nl), k), hole__0), key__)).

let R(k:bitstring) =
        new nR : bitstring;
        in (c_attack__, ynl:bitstring);
        new hole__1 : bitstring;
        out(c_attack__, enc((ynl, nR), k));
        out(c_public__, atck_enc__((enc((ynl, nR), k), hole__1), key__));
        in (c_attack__, y:bitstring);
        let (=nR, =ynl) = dec(y, k) in
        out(c_attack__, ok);
        out(c_public__, atck_enc__((ok, ok), key__)).

let FH =
        ! new k : bitstring;
        !(I(k)
        |
        R(k)).

let process__ =
        FH.

(* Modélisation des capacités de l'attaquant *)
let give_attacker_const =
        out(c_attack__, true);
        out(c_attack__, false);
        out(c_attack__, ok);
        out(c_attack__, hole).

let give_attacker_nonce =
        new n:bitstring;
        out(c_attack__, n).

let give_attacker_repl =
        in (c_attack__, x:bitstring);
        out(c_attack__, x);
        out(c_attack__, x).

let Atck_fun_cnstr_tuple_2__ =
        in (c_attack__, x__10:bitstring);
        in (c_attack__, x__11:bitstring);
        let x__12:bitstring =(x__10, x__11) in
        out(c_attack__, x__12).

let Atck_fun_destr_tuple_2__ =
        in (c_attack__, (x__8:bitstring, x__9:bitstring));
        out(c_attack__, x__8);
        out(c_attack__, x__9).
```

```
let Atck_fun_dec__ =
        in (c_attack__, x__5:bitstring);
        in (c_attack__, x__6:bitstring);
        let x__7:bitstring =dec(x__5, x__6) in
        out(c_attack__, x__7).

let Atck_fun_enc__ =
        in (c_attack__, x__2:bitstring);
        in (c_attack__, x__3:bitstring);
        let x__4:bitstring =enc(x__2, x__3) in
        out(c_attack__, x__4).

(* Envoie sur un canal publique la version réelle et la version idéalisée du message considéré *)
let p_check__ =
        in (c_public__, x__:bitstring);
        let (y__1:bitstring, y__2:bitstring) =atck_dec__(x__, key__) in
        out(c_public__, choice [y__1, y__2]).

(* Parallélisation des possibilités de l'attaquant *)
let attacker__ =
        !(give_attacker_const
        |
        give_attacker_nonce
        |
        give_attacker_repl
        |
        Atck_fun_cnstr_tuple_2__
        |
        Atck_fun_destr_tuple_2__
        |
        Atck_fun_dec__
        |
        Atck_fun_enc__).

let final_process__ =
        (process__
        |
        attacker__
        |
        phase 1;
        ! p_check__).

process final_process__
```

C Protocole de Feldhofer en Proverif testant *Frame Opacity* avec un autre codage

```
free c_public__ : channel.
```

```
free c_attack__ : channel [private].

fun monf(bitstring, bitstring) : bitstring [private].
        reduc forall x : bitstring, y : bitstring;
                fst(monf(x, y)) = x [private].
        reduc forall x : bitstring, y : bitstring;
                snd(monf(x, y)) = y [private].

free ci : channel.

free cr : channel.

free ok : bitstring.

free hole : bitstring.

fun enc(bitstring, bitstring) : bitstring.
        reduc forall xm : bitstring, xk : bitstring;
                dec(enc(xm, xk), xk) = xm.

let I(k:bitstring) =
        new nI : bitstring;
        out(c_attack__, monf(nI, nI));
        out(c_public__, monf(nI, nI));
        in (c_attack__, x1:bitstring);
        let (x:bitstring, hole__x:bitstring) = (fst(x1), snd(x1)) in
        let (=nI, xnr:bitstring) = dec(x, k) in
        new hole__0 : bitstring;
        out(c_attack__, monf(enc((xnr, nI), k), hole__0));
        out(c_public__, monf(enc((xnr, nI), k), hole__0)).

let R(k:bitstring) =
        in (c_attack__, x4:bitstring);
        let (ynI:bitstring, hole__ynI:bitstring) = (fst(x4), snd(x4)) in
        new nR : bitstring;
        new hole__2 : bitstring;
        out(c_attack__, monf(enc((ynI, nR), k), hole__2));
        out(c_public__, monf(enc((ynI, nR), k), hole__2));
        in (c_attack__, x3:bitstring);
        let (y:bitstring, hole__y:bitstring) = (fst(x3), snd(x3)) in
        let (=nR, =ynI) = dec(y, k) in
        out(c_attack__, monf(ok, ok));
        out(c_public__, monf(ok, ok)).

let FH =
        ! new k : bitstring;
        !(I(k)
        |
```

```
        R(k)).

let process__ =
FH.

let give_attacker_const =
        out(c_attack__, monf(ok, ok));
        out(c_attack__, monf(hole, hole)).

let give_attacker_nonce =
        new n:bitstring;
        out(c_attack__, monf(n, n)).

let give_attacker_repl =
        in (c_attack__, x:bitstring);
        out(c_attack__, x);
        out(c_attack__, x).

let Atck_fun_cnstr_tuple_2__ =
        in (c_attack__, x_12:bitstring);
        in (c_attack__, x_13:bitstring);
        out(c_attack__, monf((fst(x_12), fst(x_13)), (snd(x_12), snd(x_13)))).

let Atck_fun_destr_tuple_2__ =
        in (c_attack__, x11:bitstring);
        let ((x__9:bitstring, x__10:bitstring),(hole__x__9:bitstring, hole__x__10:bitstring)) =(fst(x11), snd(x11
        out(c_attack__, monf(x__9, hole__x__9));
        out(c_attack__, monf(x__10, hole__x__10)).

let Atck_fun_dec__ =
        in (c_attack__, x_7:bitstring);
        in (c_attack__, x_8:bitstring);
        out(c_attack__, monf(dec(fst(x_7), fst(x_8)), dec(snd(x_7), snd(x_8)))).

let Atck_fun_enc__ =
        in (c_attack__, x_5:bitstring);
        in (c_attack__, x_6:bitstring);
        out(c_attack__, monf(enc(fst(x_5), fst(x_6)), enc(snd(x_5), snd(x_6)))).

let p_check__ =
        in (c_public__, x__:bitstring);
        let (y__1:bitstring, y__2:bitstring) =(fst(x__), snd(x__)) in
        out(c_public__, choice [y__1, y__2]).

let attacker__ =
        !(give_attacker_const
        |
        give_attacker_nonce
        |
```

```
        give_attacker_repl
        |
        Atck_fun_cnstr_tuple_2__
        |
        Atck_fun_destr_tuple_2__
        |
        Atck_fun_dec__
        |
        Atck_fun_enc__).

let final_process__ =
        (process__
        |
        attacker__
        |
        phase 1;
        ! p_check__).

process final_process__
```