Faire garder la banque par un Coq

Judicaël Courant[1] & Jean-François Monin[2]

1: Verimag, Centre Équation
2, avenue de Vignate
F-38610 GIÈRES
`Judicael.Courant@imag.fr`
2: Verimag, Centre Équation
2, avenue de Vignate
F-38610 GIÈRES
`Jean-Francois.Monin@imag.fr`

Résumé

Nous décrivons comment Coq nous a permis de démontrer mathématiquement et formellement la sécurité d'une interface de programmation de sécurité. À notre connaissance il s'agit de la première démonstration mathématique de sécurité d'une interface de programmation. Nous présentons brièvement notre démonstration et expliquons les difficultés rencontrées lors du processus de formalisation.

1. Introduction

1.1. Contexte

Les interfaces de programmation (IP), en anglais API pour *Application Program Interface*, sont omniprésentes en programmation, qu'il s'agisse de l'interface des appels POSIX sous Unix, de la boîte à outils graphique Gtk, ... Elles permettent de définir l'accès à un composant logiciel.

Une classe particulière d'IP est celle des IP de sécurité. Celles-ci proposent des opérations de sécurité, d'identification, de signature, de chiffrement, reposant sur des mécanismes cryptographiques. L'interface de programmation de sécurité la plus connue est SSL, utilisée notamment par HTTPS (HTTP au-dessus de SSL).

Celles-ci peuvent être implantées logiciellement ou matériellement : ainsi l'IP SSL est implanté logiciellement par la bibliothèque openSSL et (partiellement) matériellement par des coprocesseurs SSL [Wik].

L'achitecture cryptographique commune (ACC) d'IBM (*Common Cryptographic Architecture*, CCA), beaucoup moins médiatique que SSL, est une IP de sécurité présente dans tous les automates bancaires. Un automate bancaire est en effet généralement composé d'un ordinateur standard (type IBM PC) auquel on a adjoint une carte PCI contenant un coprocesseur IBM 4758 implantant cette IP. L'IBM 4758 est muni d'une mémoire de petite taille, s'effaçant en cas d'attaque physique contre la carte.

Mike Bond [Bon01, Bon04] a montré que des erreurs dans la conception d'IP de sécurité permettaient d'effectuer des attaques contre des dispositifs cryptographiques : la combinaison d'appels, pourtant légaux, de façon non prévue par les concepteurs peut conduire un utilisateur à obtenir des informations qu'il était censé ignorer. Ainsi sur l'IBM 4758, on peut, en utilisant uniquement des appels corrects de l'IP, obtenir des valeurs secrètes censées rester à l'intérieur du module cryptographique.

Si l'on compare le problème de la sécurité des IP à ceux de la sécurité des protocoles cryptographiques, on peut remarquer que la modélisation est proche du modèle de Dolev-Yao dans lequel les primitives cryptographiques sont supposées parfaites et l'adversaire peut écouter, modifier et supprimer les données sur le réseau ainsi qu'effectuer des calculs hors-ligne (chiffrement et déchiffrement). La différence principale est qu'ici l'adversaire peut en outre effectuer les appels qu'il désire à l'IP avec les messages qu'il a construit ou capturés, ce qui augmente le nombre d'attaques potentielles. De plus, alors que le nombre de messages échangés dans un protocole cryptographique est relativement faible, l'attaquant peut ici conduire une attaque utilisant un nombre arbitraire d'appels consécutifs. L'analyse des IP de sécurité semble donc beaucoup plus complexe que celle des protocoles cryptographiques.

Récemment, Bond et ses collègues ont proposé d'utiliser un démonstrateur automatique pour trouver automatiquement des attaques contre une IP. Ils ont montré, sur une modélisation de l'IP ACC simplifiée, que le démonstrateur Otter [Ott], pouvait retrouver l'ensemble des failles connues de l'ACC [YAB+05].

Le problème de cette approche est, comme toute méthode de test, que la non-découverte de failles dans un temps raisonnable ne signifie pas l'absence de failles. On aimerait donc utiliser un démonstrateur automatique pour démontrer l'absence de failles. Malheureusement, alors que la preuve de l'existence d'une faille se fait en exhibant une attaque (combinaison finie d'appels de l'IP et de calculs hors-ligne révélant un secret), la démonstration de la sécurité ne peut se faire que par un raisonnement infinitaire (il s'agit de montrer qu'aucune combinaison d'appels de cette IP ne conduit à révéler un secret).

Cette difficulté nous a conduit à rejeter l'idée de faire appel à un démonstrateur complètement automatique en logique du premier ordre et à utiliser l'assistant à la démonstration Coq pour valider formellement la modélisation d'ACC proposée dans [YAB+05] que nous avons légèrement modifiée pour en corriger les failles. L'usage de Coq nous permet en effet d'effectuer des démonstrations par induction, essentielles à notre démonstration.

1.2. Notre contribution

Nos contributions sont les suivantes :

- Nous démontrons mathématiquement la sécurité d'une IP de sécurité. Effectuer une telle démonstration est considéré comme significativement plus difficile qu'exhiber une faille d'une IP, à tel point que les travaux que nous connaissons ne contiennent aucune démonstration mathématique de la sécurité d'une IP : lorsqu'il s'agit de concevoir une IP sûre, ces travaux proposent (seulement) des principes méthodologiques. À notre connaissance, notre démonstration est la première validation mathématique de sécurité d'une IP.
- La méthodologie que nous avons employée est susceptible d'être réutilisée pour d'autres systèmes.
- Nous vérifions cette démonstration dans un assistant à la démonstration. Nous réfutons ainsi la conjecture énoncée par Bond selon laquelle la preuve formelle de la sécurité d'une IP est beaucoup plus complexe que la preuve formelle d'un protocole cryptographique et requiert donc une approche formelle différente.
- La démonstration de sécurité n'a pas été trouvée *avant* la modélisation en Coq. Au contraire, cette modélisation et l'interaction avec Coq ont permis de trouver la démonstration *au fur et à mesure* de la modélisation, par une suite de conjectures, vérifiées formellement pour certaines, refutées pour d'autres lorsque le processus de validation en Coq mettait en évidence des cas problématiques. Notre travail montre donc la pertinence de l'utilisation d'un assistant à la démonstration pour *découvrir* la démonstration.
- Enfin, nous apportons quelques remarques sur l'intérêt et les difficultés que présente la vérification d'une telle démonstration avec l'assistant à la démonstration Coq.

1.3. Plan

Le plan de cet article est le suivant : partie 2 nous présentons l'IP ACC, montrons une de ses failles et expliquons quelle modification nous avons apporté à cette IP pour la corriger ; partie 3, nous expliquons notre formalisation en Coq et enfin partie 4 nous effectuons quelques remarques sur les avantages et inconvénients liés à l'utilisation de Coq.

2. L'IBM 4758 et l'interface de programmation ACC

2.1. Généralités

L'algorithme de chiffrement utilisé par l'IBM 4758 est un algorithme de chiffrement *symétrique* (le triple DES), c'est-à-dire un algorithme dans lequel la même clé est utilisé comme paramètre pour chiffrer un message puis le déchiffrer. Dans la suite nous noterons $\{x\}_k$ le résultat du chiffrement de la donnée x avec la clé k.

Pour spécifier une opération cryptographique (fournie par l'IP ou calculable hors-ligne), on utilisera par la suite la notation standard $t_1, \ldots, t_n \to t$ pour signifier que t peut être obtenu à partir du moment où l'on connaît $t_1, \ldots t_n$.

Ainsi, l'algorithme de chiffrement étant un algorithme publié, on peut facilement effectuer les opérations de chiffrement $x, k \to \{x\}_k$ et de déchiffrement $\{x\}_k, k \to x$ *hors-ligne*, c'est-à-dire sans avoir accès à un IBM 4758.

Par la suite, nous noterons $x \oplus y$ le résultat de l'opération ou exclusif bit-à-bit sur les opérandes x et y.

Afin de faciliter les comparaisons avec les travaux existants, la terminologie et les identificateurs employés sont conformes à ceux de la formalisation de Bond [YAB⁺05].

2.2. Le coprocesseur 4758

Le coprocesseur IBM 4758 est conçu pour permettre notamment de vérifier le code confidentiel associé à une carte bancaire lors d'une opération de retrait tout en gardant autant que possible confidentielles les données sensibles utilisées lors d'une opération bancaire, y compris en cas d'attaques physiques et de fraude du personnel de la banque et des programmeurs d'un automate bancaire.

L'IP de l'ACC étant assez complexe, nous travaillons sur une simplification de cette IP, celle utilisée par Bond [YAB⁺05]. Cette IP simplifiée est déjà suffisamment complexe pour posséder des failles. Nous reprenons dans la suite les noms utilisés par Bond dans son étude.

Une des fonctions de l'IBM 4758 est de permettre la vérification du code confidentiel d'une carte. Le code confidentiel *canonique* d'une carte est $\{NCC\}_P$ où NCC est le numéro de compte client du porteur (présent en clair sur la carte bancaire) et P une clé connue seulement de la banque et *identique sur tout le réseau bancaire.* Il est donc essentiel que le secret P soit bien protégé. Pour obtenir le code confidentiel *effectif* de la carte (celui que doit taper le porteur), on prend les quatre premiers chiffres $c_1, \ldots, c_4$ de la représentation *hexadécimale* du code canonique ; le code confidentiel est le nombre à quatre chiffres $\overline{d_1 d_2 d_3 d_4}$, avec $d_i = c_i \bmod 10$ pour $i = 1, \ldots, 4$ auquel on ajoute un décalage spécifié en clair sur la carte bancaire. La distinction entre le code confidentiel effectif et le code confidentiel canonique n'est pas pertinente dans la suite de notre propos, aussi nous l'ignorerons. Il est à noter que le coprocesseur propose une fonctionnalité de vérification du numéro confidentiel mais qu'en aucun cas il ne doit proposer une fonctionnalité de calcul direct du code confidentiel d'un compte arbitraire[1].

[1] Trouver le code est toujours possible mais demande beaucoup d'essais, donc de temps, donc implique une forte probabilité d'être détecté. Il s'agit donc d'un investissement risqué pour un bénéfice potentiel relativement modéré.

L'IBM 4758 a une faible mémoire : juste ce qu'il faut pour garder une clé secrète KM, générée aléatoirement à la première mise en service de la machine et appelée clé maîtresse, ainsi que pour faire quelques opérations cryptographiques.

Ainsi, la clé P n'est pas stockée dans le coprocesseur : c'est le PC qui contient le coprocesseur qui aura la charge de la stocker. Celui-ci étant considéré comme non-sûr, on la stockera sous la forme $\{P\}_{KM}$. Ainsi, si l'on suppose le chiffrement parfait, seule la connaissance de KM situé dans le coprocesseur sécurisé permet de recouvrer P. Plus précisément, les valeurs chiffrées qui sont stockées dans le PC sont étiquetées avec une information de typage : ainsi on ne stocke par $\{P\}_{KM}$ mais $\{P\}_{PIN \oplus KM}$, où PIN est un entier, publiquement connu, représentant le type des clés permettant d'obtenir le code confidentiel à partir du numéro de compte. On utilisera par la suite l'abus de langage consistant à dire que $\{P\}_{PIN \oplus KM}$ est la donnée P chiffrée par KM dans le type PIN.

La difficulté majeure de mise en place d'un automate bancaire réside dans le chargement des secrets bancaires comme la clé P. Pour transmettre ces secrets de la banque à l'automate, on va mettre en place un canal de communication sécurisé : il s'agit simplement d'un canal de communication classique, sur lequel on chiffre les données avec une clé de communication KEK (Key Encrypting Key, clé de chiffrement de clé) connue seulement du coprocesseur et du serveur de la banque avec lequel il communique. Le problème est donc simplement de faire que le coprocesseur et le serveur de la banque partagent ce secret KEK. Dans ce but, le serveur de la banque génère une clé secrète KEK, que l'on partage en trois secrets K_1, K_2, K_3 tels que $KEK = K_1 \oplus K_2 \oplus K_3$, qui sont transmis successivement par trois personnes de confiance (de préférence différentes) et entrés manuellement dans le coprocesseur. Reconstituer KEK pour un attaquant nécessite donc d'intercepter chacun des trois courriers K_1, K_2 et K_3. Le coprocesseur calcule quant à lui la valeur $K_1 \oplus K_2 \oplus K_3$ ce qui lui permet d'obtenir KEK. Le secret KEK est donc partagé entre le serveur de la banque et le coprocesseur.

Toujours pour des raisons de place, KEK ne sera pas stocké sur le coprocesseur mais sur le PC sous la forme $\{KEK\}_{IMP \oplus KM}$ où IMP est le type des clés d'importations. De même les valeurs intermédiaires K_1 et $K_1 \oplus K_2$ ne sont pas stockées en clair. Le coprocesseur propose en effet les deux opérations suivantes pour stocker des morceaux de clé :

$$x, y, \{z\}_{x \oplus KP \oplus KM} \to \{z \oplus y\}_{x \oplus KP \oplus KM} \tag{1}$$

$$x, y, \{z\}_{x \oplus KP \oplus KM} \to \{z \oplus y\}_{x \oplus KM} \tag{2}$$

Ces deux opérations utilisent des données chiffrées dans le type $x \oplus KP$ qu'il faut comprendre comme le type des morceaux de clés de type x (KP peut ici être vu comme un type paramétré, le type des morceaux de clés). La première permet d'ajouter une valeur y à une partie de clé z pour obtenir une nouvelle partie de clé $y \oplus z$; la seconde, avec les mêmes entrées, produit quant à elle une clé complète.

Une opération du coprocesseur est prévue pour permettre d'importer une valeur x chiffrée par une clé d'importation :

$$t, \{k\}_{IMP \oplus KM}, \{x\}_{t \oplus k} \to \{x\}_{t \oplus KM} \tag{3}$$

Autrement dit : étant donnée une valeur x chiffrée par k dans le type t et le chiffrement de k par KM dans le type IMP, on peut obtenir le chiffrement de x par KM dans le type t.

Le secret P sera donc transmis sous la forme $\{P\}_{PIN \oplus KEK}$, ce qui permettra au PC d'obtenir la valeur $\{P\}_{PIN \oplus KM}$.

Le coprocesseur possède une clé secrète *EXP₁* typée dans le type EXP des clés d'exportations, de façon symétrique aux clés d'importations. Une opération du coprocesseur permet d'exporter une valeur x chiffrée par une clé d'exportation :

$$t, \{k\}_{EXP \oplus KM}, \{x\}_{t \oplus KM} \to \{x\}_{t \oplus k} \tag{4}$$

Cette règle n'intervient pas dans les attaques mentionnées dans [YAB+05]. Elle y est mentionnée pour montrer que leur outil trouve des failles même si on ajoute plus de règles que nécessaire pour cela.

Nous verrons par la suite, paragraphe 3.2.4, que cette règle nous a conduit à une définition plus fine des termes «sensibles» que ce que nous pensions au départ. Enfin, [YAB+05] ajoute également une clé KEK_2 typée à la fois dans les clés d'importation et les clés d'exportations. Cet ajout ne perturbe pas leur recherche de preuve, il ne modifie pas non plus notre développement.

Le 4758 propose également la possibilité au PC de chiffrer ses données applicatives pour ses propres besoins : lorsqu'une application veut chiffrer une donnée x, elle génère une clé k. Pour ne pas garder celle-ci dans sa propre mémoire, elle utilise un appel $k \to \{k\}_{DATA \oplus KM}$ permettant de chiffrer sa clé par KM dans le type $DATA$ des données utilisateurs. L'ACC propose alors deux appels

$$x, \{k\}_{DATA \oplus KM} \to \{x\}_k \tag{5}$$

et

$$\{x\}_k, \{k\}_{DATA \oplus KM} \to x \tag{6}$$

permettant respectivement de chiffrer et déchiffrer des données avec la clé k. Contrairement aux autres appels présentés jusqu'ici, l'idée n'est pas de se prémunir contre un comportement frauduleux du PC, mais de prémunir les données du PC contre un attaquant extérieur qui arriverait à lire la mémoire du PC, par exemple par des mesures physiques extérieures (rayonnement électromagnétique, perturbations sur la ligne électrique, ...).

2.3. Les failles de l'IP ACC

Plusieurs failles ont été détectées dans l'IP de sécurité ACC [Bon04]. La plus simple consiste à faire importer la clé KEK sous $DATA$ au lieu du type IMP. De la sorte, une application maligne peut ensuite facilement déchiffrer toutes les informations reçues par le coprocesseur. Pour effectuer cet import, l'attaquant procède de la façon suivante : lorsque le dernier morceau K_3 de la clé KEK est donné, il intercepte K_3 et au lieu de se contenter d'appliquer l'opération (2) d'intégration du dernier morceau de clé

$$IMP, K_3, \{K_1 \oplus K_2\}_{IMP \oplus KP \oplus KM} \to \{K_1 \oplus K_2 \oplus K_3\}_{IMP \oplus KM}$$

il effectue également cette opération avec $K_3 \oplus PIN \oplus DATA$:

$$IMP, K_3 \oplus PIN \oplus DATA, \{K_1 \oplus K_2\}_{IMP \oplus KP \oplus KM} \to \{K_1 \oplus K_2 \oplus K_3 \oplus PIN \oplus DATA\}_{IMP \oplus KM}$$

il a donc obtenu $\{KEK\}_{IMP \oplus KM}$ et $\{KEK \oplus PIN \oplus DATA\}_{IMP \oplus KM}$. Dès lors, lorsqu'il reçoit $\{P\}_{PIN \oplus KEK}$, la règle (3) peut être utilisée pour importer $x = P$ non seulement avec le type $t = PIN$ et $k = KEK$ – comme prévu – mais aussi comme avec le type $t = DATA$, en prenant $k = KEK \oplus PIN \oplus DATA$, puisque $PIN \oplus KEK = DATA \oplus (KEK \oplus PIN \oplus DATA)$.

On obtient ainsi $\{P\}_{DATA \oplus KM}$, ce qui permet, par la règle (5) de chiffrement de données applicatives, de calculer $\{NCC\}_P$ pour n'importe quel compte.

En interceptant seulement le troisième morceau de clé (K_3), l'attaquant a donc réussi à obtenir des informations sensibles, alors que l'IP ACC était censée résister à l'interception de deux quelconques morceaux de clé parmi K_1, K_2 et K_3.

Les conséquences de cette faille doivent cependant être relativisées dans la mesure où l'IP ACC gère des notions d'autorisations : tout le monde n'est pas autorisé à effectuer toutes les opérations. Seule une gestion insuffisamment rigoureuse de ces utilisations pouvait conduire à l'attaque décrite ci-dessus. La correction de cette faille a essentiellement consisté à corriger le manuel de référence du 4758 pour expliquer comment gérer ces autorisations et à en avertir les banques. Bond suggère qu'une solution à plus long terme serait d'utiliser une fonction à sens unique plutôt que la fonction ou exclusif (voir [Bon01], paragraphe 5.1, p. 230) mais à notre connaissance ne donne pas de démonstration de la sécurité de l'API ainsi corrigée.

La seule différence entre notre API et celle présentée dans [YAB+05] est l'utilisation d'une telle fonction à sens unique H, prenant deux arguments : un entier représentant un type et un représentant une clé. On suppose cette fonction parfaite, c'est-à-dire que la donnée de $H(x, y)$ ne permet pas de calculer x ni y et que l'on ne peut pas en pratique exhiber (x_1, y_1) et (x_2, y_2) distincts vérifiant $H(x_1, y_1) = H(x_2, y_2)$.

3. Formalisation

L'idée générale de la formalisation est la suivante. On définit inductivement le prédicat `known` des valeurs connues, c'est-à-dire de celle que l'on connaît au départ ou qu'on peut obtenir à partir de valeurs connues en leur appliquant des opérations calculables comme $\oplus$, H, le chiffrement, ou des opérations effectuées par le coprocesseur cryptographique 4758. Notre modélisation s'intéresse au cas où K_1 et K_2 n'ont pas été interceptés mais où K_3 l'a été. Nous supposons donc que nous avons `known`(K_3). En outre, au lieu d'introduire K_1, K_2 et K_3, et de manipuler l'expression $K_1 \oplus K_2 \oplus K_3$, nous avons adopté l'astuce de [YAB+05] consistant à noter KEK la somme $K_1 \oplus K_2 \oplus K_3$. On remarque alors que K_1 et K_2 n'interviennent dans la modélisation que sous la forme $K_1 \oplus K_2$ ou $K_1 \oplus K_2 \oplus K_3$, qu'on peut écrire respectivement $KEK \oplus K_3$ et KEK, ce qui permet de supprimer toute référence à K_1 et K_2, donc de les supprimer de la modélisation.

De façon similaire au modèle de Dolev-Yao pour les protocoles cryptographiques, nous supposons en outre que ces valeurs connues sont les seuls termes dont peut avoir connaissance un utilisateur de l'IP, honnête ou non. Nous excluons donc implicitement de notre étude toutes les attaques qui s'appuieraient sur des attaques des primitives cryptographiques elles-mêmes, telles que les attaques consistant à trouver la clé par laquelle est chiffrée un (ensemble de) message(s) chiffré(s) connu(s) (dont on connaît éventuellement ausi les textes clairs correspondants), à inverser la fonction de hachage sur certaines valeurs ou à trouver des collisions sur la fonction de hachage (c'est-à-dire deux couples distincts (x_1, y_1) et (x_2, y_2) vérifiant $H(x_1, y_1) = H(x_2, y_2)$). Autrement dit, nous supposons que ces attaques sont trop difficiles à conduire (en terme de puissance de calcul nécessaire) pour un utilisateur malhonnête.

L'objectif est alors de démontrer que, parmi les valeurs connues, on ne trouve ni KEK, ni KM, ni P, ni $\{NCC\}_P$. Un simple examen des différents cas pour `known` ne suffit pas, car certains sous-cas en hypothèse font apparaître des termes plus grands que ceux de la conclusion (par exemple `K_decrypt` ci-après). On procède en introduisant une sur-approximation de `known` ayant cette fois de bonnes propriétés structurelles : `unc`, qui désigne les termes *déclassifiés*, au sens militaire du terme, c'est-à-dire dont on estime qu'il n'y a pas de danger pour la sécurité à ce qu'ils soient rendus publics. Nous verrons que la définition précise de ce prédicat est plus délicate qu'il n'y paraît. C'est l'utilisation de l'assistant de preuve qui a permis de mettre en évidence les insuffisances des premiers jets et de concevoir une version utilisable de `unc`, permettant de démontrer effectivement que, dans notre modèle, tout terme calculable par l'utilisateur est déclassifié.

Une des principales difficultés techniques concerne la gestion de l'opérateur $\oplus$: il devient nécessaire de raisonner modulo l'associativité, la commutativité et l'involutivité de cet opérateur. Ces propriétés algébriques jouent d'ailleurs un rôle essentiel en cryptographie, à la fois positif et négatif : dans le cas d'étude considéré, elles sont exploitées pour transmettre KEK en trois parties, mais laissent aussi la place à des scénarios indésirables. Plutôt que raisonner sur des classes d'équivalences au moyen de setoïdes, nous avons choisi d'utiliser des représentants canoniques. Nous procédons en deux phases :

- le cœur de la démonstration s'effectue dans le contexte d'une fonction de normalisation abstraite `norm` ;
- on construit par ailleurs une telle fonction en utilisant un ordre bien fondé entre les termes d'une même somme.

Dans ce qui suit nous n'évoquons, parmi les éléments relatifs au processeur 4758, que ceux qui ont été mentionnés dans les sections précédentes. Le développement complet en comprend quelques autres.

3.1. Énoncés principaux

3.1.1. Définitions

On se donne tout d'abord le type `secret_const` des constantes secrètes.

```
Inductive secret_const : Set :=
  | KM : secret_const
  | KEK : secret_const
  | P : secret_const
  | KEK2 : secret_const
  | EXP1 : secret_const
```

De manière analogue, on introduit le type `public_const` des constantes publiques, qui contient notamment *DATA*, *PIN*, *IMP*, K_3, *NCC* et *KP*. On peut alors définir le type des termes comme suit :

```
Inductive term : Set :=
  | Zero : term
  | PC :> public_const -> term
  | SC :> secret_const -> term
  | E : term -> term -> term       (* chiffrement *)
  | Xor : term -> term -> term     (* ou exclusif bit-à-bit *)
  | Hash : term -> term -> term.   (* fonction à sens unique *)
```

Avec les notations suivantes :

```
Notation "{. x } y " := (E x y) (at level 0).
Notation " x +* y " := (Hash x y) (at level 20, left associativity).
Notation " x ⊕ y " := (Xor x y).
```

Les termes calculables sont définis comme suit.

```
Inductive init_known : term -> Prop :=
  | K_Zero : init_known Zero
  | K_PC : ∀ c : public_const, init_known c
  | P_KEK_PIN : init_known {.P }(PIN +* KEK)
  | K3_KEK : init_known {.norm (KEK ⊕ K3)}(IMP +* KP +* KM)
  | KEK2_IMP : init_known {.KEK2 }(IMP +* KP +* KM)
  | KEK2_EXP : init_known {.KEK2 }(EXP +* KP +* KM)
  | EXP1_EXP : init_known {.EXP1 }(EXP +* KM)
.

Inductive known : term -> Prop :=
  | K_init : ∀ x : term, init_known x -> known x
  | K_Xor : ∀ x y : term, known x -> known y -> known (x ⊕ y)
  | K_Hash : ∀ x y : term, known x -> known y -> known (x +* y)
  | K_E : ∀ x k : term, known x -> known k -> known {.x}k
  | K_decrypt : ∀ x k : term, known {.x}k -> known k -> known x
```

```
  | K_key_import :
      ∀ x y z, known x ->
      known {.z}(IMP ++ KM) -> known {.y}(x ++ z) -> known {.y}(x ++ KM)
  | K_key_part_import_completing :
      ∀ x y z, known x -> known y ->
      known {.z}(x ++ KP ++ KM) -> known {.z ⊕ y}(x ++ KM)
  | K_key_part_import_notcompleting :
      ∀ x y z, known x -> known y ->
      known {.z}(x ++ KP ++ KM) -> known {.z ⊕ y}(x ++ KP ++ KM)
  | K_encrypt_using_data_key :
      ∀ x y, known x -> known {.y}(DATA ++ KM) -> known {.x}y
  | K_key_export :
      ∀ x y z, known x -> known {.y}(x ++ KM) ->
      known {.z}(EXP ++ KM) -> known {.y}(x ++ z)
  | K_decrypt_using_data_key :
      ∀ x y, known {.x}y -> known {.y}(DATA ++ KM) -> known x
  | K_Eq : ∀ x y : term, known x -> Eq x y -> known y
.
```

Les termes calculables par l'utilisateur sont indiqués par les règles `K_init`, `K_Xor` et `K_Hash`, `K_E` et `K_decrypt` (l'algorithme de cryptage est supposé connu), les autres règles font appel à l'API du coprocesseur 4758 modifié, à l'exception de la dernière qui permet de raisonner modulo l'égalité des termes (congruence engendrée par les règles sur l'opérateur $\oplus$). Le tableau suivant explicite la correspondance entre les constructeurs Coq et les numéros des opérations présentées précédemment. Comme annoncé, on a remplacé l'emploi de $\oplus$ par la fonction de hachage en plusieurs endroits.

Nom du constructeur Coq	Numéro de formule correspondant
`K_key_import`	(3)
`K_key_export`	(4)
`K_key_part_import_completing`	(2)
`K_key_part_import_notcompleting`	(1)
`K_encrypt_using_data_key`	(5)
`K_decrypt_using_data_key`	(6)

Enfin, on définit les termes interdits.

```
Inductive forbidden : term -> Prop :=
  | F_KEK : forbidden KEK
  | F_KM : forbidden KM
  | F_P : forbidden P
  | F_E_ACC_P : forbidden {.NCC }P.
```

3.1.2. Théorème

Théorème 1 *Aucun terme interdit n'est calculable.*

Formellement :

```
Theorem security_results : ∀ x, forbidden x -> ¬ known x.
```

3.2. Démonstration

3.2.1. Contexte : normalisation axiomatique

On se donne ici une fonction de normalisation sur les termes notée `norm`. Deux termes sont équivalents si leurs normes sont égales, ce qui signifie qu'ils sont égaux modulo les propriétés algébriques de $\oplus$. En particulier, la fonction `norm` élimine les doublons de la forme $\ldots x \oplus \ldots \oplus x \ldots$ (et plus généralement ceux de la forme $\ldots x \oplus \ldots \oplus x' \ldots$ où x et x' ont même norme). Le prédicat `is_nf` indique qu'un terme est en forme normale.

3.2.2. Déclassification d'un terme

Un terme déclassifié est défini comme un terme dont la norme verifie le prédicat `unc_nf` défini inductivement comme suit. Deux autres prédicats auxiliaires sont nécessaires :

- la relation `sub_xor`(t_1, t_2), qui indique qu'un terme t_1 non construit par $\oplus$ fait partie d'une séquence de $\oplus$, le terme t_2 ; nous ne le détaillons pas ici ;
- Comme noté au paragraphe 2.2, les données chiffrées du type *DATA* ne recouvrent pas la même problématique de confidentialité que les autres données chiffrées par le coprocesseur. Il faudra donc distinguer à certains moment le type *DATA* et les autres types. Plus précisément, au cours de la preuve il est apparu nécessaire de généraliser cette distinction entre les types qui contiennent *DATA* (c'est-à-dire *DATA* et les types de la forme "morceau de ... morceau de *DATA*") et les autres. Nous détaillons ce problème au paragraphe 3.2.4. Pour cela, on définit le prédicat `contains_data` :

```
Inductive contains_data : term -> Prop :=
  | CD_Data : contains_data DATA
  | CD_Hash : ∀ x, contains_data x -> contains_data (x  +* KP).

Inductive unc_nf : term -> Prop :=
  | U_Zero : unc_nf Zero
  | U_PC   : ∀ c, unc_nf (PC c)
  | U_E    : ∀ x y, unc_nf x -> unc_nf y -> unc_nf {.x}y
  | U_Data : ∀ x y z, unc_nf x -> is_nf y -> contains_data y ->
               is_nf z -> unc_nf {.x}(y +* z)
  | U_Xor  : ∀ x y, unc_nf x -> unc_nf y -> is_nf (x ⊕ y) -> unc_nf (x ⊕ y)
  | U_Hash : ∀ x y, unc_nf x -> unc_nf y -> unc_nf (x +* y)
  | U_crpt : ∀ x y c1 c2 t, is_nf x -> is_nf y -> is_nf t ->
             sub_xor (SC c1) x -> sub_xor (SC c2) y -> ¬contains_data t ->
             unc_nf {.x}(t +* y).

Definition unc x := unc_nf (norm x).
```

Les clauses pour `unc_nf` sont pour la plupart assez naturelles, mais celle qui concerne l'encryption simple (`U_E`) demande quelques explications. Nous y revenons au paragraphe 3.2.4.

3.2.3. Lemme principal

Lemme 1 *Un terme calculable est nécessairement déclassifié.*

Formellement :

```
Theorem main : ∀ x, known x -> unc x.
```

Le théorème 1 s'obtient en corollaire : aucune des clauses de `unc_nf` n'autorise l'un des quatre termes interdits (sauf `U_E` en ce qui concerne $\{NCC\}_P$ car `U_E` peut s'appliquer mais cela se ramène à obtenir P, ce qui est impossible).

La démonstration du lemme 1 se fait par récurrence structurelle sur la définition de `known`. Cela revient à montrer que chaque clause de `known` conserve le prédicat `unc`. Par exemple, pour la clause `K_E`, on doit démontrer le lemme suivant, qui se ramène simplement à `U_E` après expansion des définitions et normalisation :

```
Lemma unc_e : ∀ x y, unc x -> unc y -> unc {.x}y.
```

3.2.4. Déclassification d'un terme simplement crypté

On pourrait imaginer, ainsi que nous avions essayé dans une première version de `U_E`, qu'il suffit que x soit déclassifié pour que $\{x\}_y$ le soit. Il s'avère en fait qu'une telle possibilité pourrait permettre d'extraire des informations interdites. En effet, la règle d'export (règle (4)) nous permet de choisir pour clé d'exportation tout terme x chiffré avec EXP+*KM. Si x est une clé d'exportation, on peut exporter toute donnée secrète z de type t sous la forme $\{z\}_{t\texttt{+*}x}$. Si x est connu, on peut alors connaître z. Une clé connue ne doit donc jamais pouvoir être utilisée comme clé d'exportation car cela permettrait de connaître toutes les clés secrètes contenues dans le coprocesseur. Si x est connu, il ne faut donc surtout pas déclassifier $\{x\}_{EXP\texttt{+*}KM}$.

Cela nous conduit aux remarques et aux règles suivantes :

- Si x est déclassifié, il faut vérifier qu'aucun mauvais usage ne pourrait être fait de la connaissance de $\{x\}_y$ avant de déclassifier celui-ci.
- $\{x\}_y$ peut être déclassifié à partir du moment où x et y le sont déjà, car l'utilisateur peut déjà calculer $\{x\}_y$ à partir de x et y par ses propres moyens (règle `U_E`).
- $\{x\}_{DATA\texttt{+*}KM}$ peut être déclassifié si x est déclassifié car les données chiffrées dans le type $DATA$ n'ont aucun usage sensible du point de vue de la sécurité des secrets du coprocesseur : le type $DATA$ correspond uniquement à des données ou des clés de données applicatives (règle `U_Data`).
- À l'inverse, il serait sans doute imprudent de déclassifier une donnée $\{x\}_{DATA\texttt{+*}KM}$ si x est censé rester secret.
- Il s'agit cependant de déclassifier les données qui vont devoir sortir du coprocesseur. On acceptera donc de déclassifier les termes $\{x\}_{t\texttt{+*}y}$ typés dans t dans la mesure où x est secret (ce qu'on peut constater par la présence d'une constante secrète dans le terme), où y est secret (même chose) et où t n'est pas $DATA$ (règle `U_crpt`).
- Plus précisément, la règle d'import de morceaux de clés (2) permet de faire passer une donnée du type t+*KP au type t. Donc si l'utilisateur à accès à un terme $\{x\}_{t\texttt{+*}KP\texttt{+*}y}$, il va pouvoir obtenir $\{x\}_{t\texttt{+*}KP}$. Donc dans le point précédent, il faut non seulement interdire à t de valoir $DATA$, mais également lui interdire de valoir $DATA$+*KP, mais aussi $DATA$+*KP+*KP car on pourrait se ramener à $DATA$+*KP, ... Le rôle du prédicat `contains_data` est précisément d'effectuer cette vérification. Nous avons également utilisé ce prédicat pour généraliser la règle de déclassification `U_Data`, bien que cela ne soit probablement pas nécessaire, en l'absence de règle permettant de construire des morceaux de clés à partir d'une clé.

3.3. Normalisation de termes

Chaque terme peut être vu comme une alternance de couches entre des constructeurs `Xor` ou différents de `Xor`. Normaliser un terme va consister à l'aplatir, (en usant de l'associativité de $\oplus$), puis à trier la liste obtenue (en utilisant la commutativité). Le tri s'effectue suivant un ordre bien fondé impliquant

- une notion ad-hoc de poids

– un ordre lexicographique pour les termes de même poids. On démontre que l'ensemble des termes de poids majoré par une valeur donnée est fini, et que toute relation irréflexive et transitive sur un ensemble fini est bien fondée.

Au passage, deux termes identiques vont devenir contigüs et seront supprimés par involutivité de $\oplus$.

Au moment de la soumission de cet article, le lien entre cette partie du développement et la présentation axiomatique de la fonction de normalisation n'est pas encore terminé. La vérification de notre développement n'est donc pas complète, une erreur dans les axiomes de la fonction de normalisation étant potentiellement susceptible de mettre en cause la validité de nos résultats. Cependant, nous sommes relativement confiants sur ce point :

– D'une part, nous avons réduit la complexité du problème initial à celle de la normalisation d'une récriture sur les termes. Il est peu probable qu'une erreur se soit glissée dans l'axiomatisation de cette normalisation, puisqu'il s'agit essentiellement de termes avec un opérateur associatif-commutatif (xor), un élément neutre vis-à-vis de cet opérateur et des symboles de fonctions non-interprétés (hachage et chiffrement). La difficulté de formalisation vient plus de l'absence d'outils Coq pour gérer une telle récriture que de l'originalité de cette relation de récriture.
– D'autre part, l'avantage d'un outil interactif comme Coq est que l'utilisateur peut voir comment et où sont utilisés les axiomes qu'il a déclarés. Aucune de ces utilisations ne nous a semblée suspecte. Autrement dit, même si l'axiomatisation de la fonction de normalisation comportait une erreur, nous estimons qu'il serait relativement aisé de corriger celle-ci tout en préservant l'essentiel de nos preuves. À l'inverse, dans un outil de preuve complètement automatique, une incohérence dans l'axiomatique d'un développement peut conduire l'outil à dériver la proposition absurde puis à utiliser cette proposition absurde pour montrer toutes les propositions de l'utilisateur, sans que celui-ci ne se rende compte de la raison pour laquelle l'outil valide ses propositions.

4. Remarques

Nous procédons maintenant à quelques remarques sur le processus de formalisation en Coq lui-même.

Les auteurs de cet article connaissent bien Coq, mais ne l'avaient pas utilisé pour faire des démonstrations demandant un emploi significatif de tactiques depuis longtemps (1993 pour le premier, 1999 pour le second). L'ensemble de tactiques disponibles a profondément évolué et permet à l'évidence d'effectuer des preuves beaucoup plus rapidement.

Nous passons en revue la "boîte à outils" que propose Coq pour effectuer des démonstrations, classée par les différents domaines de nos démonstrations.

4.1. Tactiques d'usage général

La tactique `assert` permet d'effectuer des preuves avant et donne des preuves plus lisibles qu'une succession d'`elim`, `intros` et `apply`. La tactique cut offrait déjà des bénéfices similaires, mais il est nettement plus naturel d'une part de pouvoir directement donner un nom à un sous-lemme énoncé par `assert` et d'autre part d'effectuer la démonstration du sous-lemme avant celle de la proposition principale.

Régulièrement, on éprouve le besoin d'appliquer à l'ensemble des hypothèses du but ou à une hypothèse H particulière une tactique qui n'est prévue que pour s'appliquer à la conclusion du but. Par exemple les tactiques `autorewrite`, `ring`. Il est dommage de ne pas avoir une tactique d'ordre supérieur `onhyp`, qui prendrait en argument une tactique *tac* et un nom d'hypothèse H et qui permettrait de faire simplement la séquence classique `generalize` H `;` `clear` H `;` *tac* `;` `intro` H.

De même il serait intéressant d'avoir une tactique `onallhyps` permettant d'appliquer une tactique à tous les buts.

4.2. Arithmétique

La tactique `omega` permet résoudre des buts solubles dans l'arithmétique de Presburger. Elle est explicitement utilisée 34 fois dans notre développement au cours de la démonstration de 24 résultats (Lemma, Theorem, Remark) différents.

Les limitations d'`omega` sont de deux ordres.

La première est purement technique et vient de la façon dont sont reliées les propositions de l'arithmétique de Presburger et les buts Coq : `omega` parcours l'ensemble des hypothèses dont il extrait les hypothèses $H_1, \ldots, H_n$ qu'il reconnaît comme étant des hypothèses de l'arithmétique de Presburger, vérifie que la conclusion C du but est une proposition de l'arithmétique de Presburger (`False` en fait partie), puis résoud la proposition $H_1 \wedge \ldots \wedge H_n \rightarrow C$. Si C ne fait pas partie de l'arithmétique de Presburger, `omega` échoue immédiatement. Nous avons rencontré des buts où la conclusion n'était pas dans l'arithmétique de Presburger mais où les hypothèses étaient absurdes : `omega` refusait alors de démontrer le but alors que le remplacement de la conclusion par `False` grâce à un `elimtype False` suffisait pour qu'`omega` résolve le but. Pour lever cette limitation, il suffirait qu'`omega` traduise un but Coq en la proposition $H_1, \ldots, H_n \rightarrow C'$ où C' est la conclusion du but Coq si celle-ci est dans l'arithmétique de Presburger et `False` sinon.

La seconde limitation est celle de l'arithmétique de Presburger par rapport à l'arithmétique dans sa généralité. Ainsi, pour résoudre un simple but tel que $x > 0 \rightarrow y > 0 \rightarrow x \times y > 0$ avec x et y dans **nat** ou dans $\mathbf{Z}$, `omega` n'est d'aucun secours. En ce qui concerne ce but particulier, un simple `auto with *` le résout. Les autres buts non solubles par `omega` correspondaient à du raisonnement équationnel sur l'anneau $\mathbf{Z}$. Nous avons alors utilisé la tactique `ring` : celle-ci permet de normaliser des termes dans un anneau en les développant et en les écrivant comme somme de monômes à coefficients dans $\mathbf{Z}$, en choisissant un ordre canonique pour les variables constituant chaque monôme. Sur trois utilisations de `ring`, nous avons une utilisation servant à normaliser un sous-terme du but avant de rendre celui-ci trivial (réflexivité) et deux servant à permettre simplement à `omega` d'identifier deux termes. Les points suivants nous ont compliqué la tâche :

- sauf dans le cas d'une égalité dont les deux membres sont les termes à normaliser, `ring` ne choisit pas lui-même les termes à normaliser. Ainsi, pour un but de la forme $n + n_1 - n_1 <= n$, l'appel de `ring` sans argument échoue. On peut envisager d'écrire des tactiques dans Ltac au cas par cas pour pallier cette insuffisance mais il serait certainement préférable que `ring` traite l'ensemble de ces cas.
- `ring` ne sait pas récrire dans une hypothèse.
- l'action de `ring` n'est pas idempotente, car l'ordre sur les variables choisi par l'appel de `ring` dépend en fait de l'ordre dans lequel ces variables apparaissent dans la conclusion du but courant ou de ses arguments. Ainsi, `ring` récrit le but $x = x + y$ en $x = y + x$ et vice-versa. Ce comportement rend plus délicate l'écriture de tactiques dans Ltac pour normaliser la conclusion et l'ensemble des hypothèses d'un but.

4.3. Raisonnement équationnel

Trois tactiques nous été extrêmement utiles pour effectuer du raisonnement équationnel :

- `subst` permettant de substituer toutes les occurences d'une variable x dans un but s'il existe une hypothèse $x = t$ où x est une variable.
- `autorewrite` qui nous a permis d'utiliser de façon calculatoire la présentation axiomatique de notre fonction de normalisation des termes `norm`. Nous regrettons cependant qu'`autorewrite` ne puisse être utilisée sur les hypothèses.

– `congruence` qui permet de démontrer des égalités par calcul de la clôture congruente d'égalités, notamment, lors de raisonnement impliquant des valeurs supposées en forme normale : une égalité de la forme $x = \mathtt{norm}\ x$ ne peut en effet pas être éliminée par `subst` (le membre droit de l'égalité faisant référence à x). Cette tactique ne sait malheureusement pas résoudre des buts de la forme $H1 : x = y, H2 : Px \vdash Py$, alors que cela s'avérerait utile.
– `discriminate` et `injection`.

4.4. Ltac

Les tactiques que nous avons cependant le plus utilisées sont des tactiques que nous avons nous-mêmes écrites en Ltac. Indéniablement, la possibilité pour l'utilisateur d'écrire lui-même ses tactiques est une amélioration majeure du système Coq. Nous avons ainsi écrit une tactique `PROGRESSO` qui applique sur le but courant une tactique parmi un ensemble de tactiques disponibles. Nous avons choisi de nous restreindre à des tactiques parmi les trois catégories suivantes :
– élimination des hypothèses inutiles ($t = t$, hypothèses en double, ...)
– progression du but : application de tactiques qui transforment un but prouvable en un autre but prouvable (et un seul), de préférence plus simple. Parmi celles-ci, on trouve `subst`, `injection`, `simpl`, `intro`. Nous avons également utilisé un minimum de raisonnement propositionnel : élimination des conjonctions, application du *modus ponens*.
– résolution du but : `omega`, `elimtype False ; omega`, `solve [intuition (eauto with *)]`

Par ailleurs, on impose aussi que les différentes tactiques échouent quand elle ne résolvent pas le but et qu'elles ne le font pas progresser. Ainsi, on peut employer `repeat PROGRESSO` pour simplifier le but, voire le résoudre.

Nous avons étendu cette tactique au cours de notre développement pour résoudre certains buts spécifiques à notre développement. Ainsi, nous avons ajouté une utilisation d'`autorewrite` pour normaliser les expressions du but suivant les axiomes donnés pour notre fonction de normalisation des termes.

La tactique `PROGRESSO` et les quatre tactiques que nous en avons dérivées sont utilisées 58 fois dans notre développement, quasiment toujours en argument de la tactique d'ordre supérieur `repeat` (à rapprocher de 189 fois pour `intuition`, `auto` et `eauto` réunies).

Alors qu'il est parfois difficile au cours d'une démonstration de comprendre quels sont les buts réellement pertinents parmi ceux qu'a généré Coq et quels sont les buts triviaux, cette famille de tactiques permet d'éliminer très facilement les buts inintéressants, c'est-à-dire ceux qui sont suffisamment communs et faciles pour qu'on ait décrit en Ltac leur résolution.

L'utilisation de ces tactiques permet de faire de la preuve automatique spécifique au domaine que l'on modélise. Ce type de démonstration peut en partie être conduit avec `auto`, en utilisant les possibilités de déclaration d'indications (commande `Hint`) mais alors qu'`auto` n'applique des tactiques que de façon dirigée par la tête de la conclusion du but, nous utilisons ici d'une part des tactiques dirigées par le but et d'autres dirigées par les hypothèses. En un sens, le fonctionnement de `PROGRESSO` est plus proche de celui d'`intuition`, mais les possibilités de paramétrage de cette dernière tactique sont malheureusement encore presque inexistantes.

La tactique `PROGRESSO` est programmée de façon très naïve, mais les performances restent en général acceptables : en effet, le temps d'exécution d'une commande `PROGRESSO` est négligeable pour l'utilisateur et si le temps mis par `PROGRESSO` est supérieur à une seconde (ce qui arrive rarement lors de la résolution d'un unique but), on sait que cela signifie que le but va être, sinon résolu, du moins bien nettoyé, chaque application réussie de `PROGRESSO` contribuant à simplifier le but.

Il serait intéressant de proposer des mécanismes génériques en Coq permettant d'enregistrer des tactiques, en les classant dans les catégories suivantes :
– les tactiques de simplification du but (qui suppriment les hypothèses inutiles) ;

– les tactiques faisant progresser le but (qui en préservent le caractère prouvable, et ne multiplient pas le nombre de buts) ;
– les autres tactiques inversibles (qui préservent le caractère prouvable du but et multiplient potentiellement le nombre de buts) ;
– les tactiques de recherche de preuve, dont l'application ne préserve pas le caractère prouvable du but (c'est l'idée des `Hint Resolve` d'`auto`).

Coq pourrait alors fournir quatre tactiques génériques :
– une tactique appliquant uniquement les simplifications ;
– une tactique appliquant de plus les tactiques de progression ;
– une tactique appliquant en outre les autres tactiques inversibles (similaire à un `intuition idtac`) ;
– une tactique effectuant de surcroît des opérations potentiellement non inversibles, et revenant en arrière lorsque celles-ci ne résolvent pas le but (similaire à un `intuition` mais avec entrelacement des étapes de recherche non inversibles et des étapes propositionnelles).

5. Conclusion et perspectives

Coq nous a permis d'obtenir une démonstration de sécurité d'une IP de sécurité pour une formalisation de l'IP ACC. Les sources Coq de notre formalisation peuvent être téléchargés à l'adresse `http://www-verimag.imag.fr/~monin/Proof/API_defense/`.

Les axes de travaux futurs sont les suivants :
– effectuer une modélisation plus réaliste de l'IP ACC ;
– modéliser d'autres IP de sécurité ([Bon04] en présente plusieurs, notamment le *Visa Security Module*) ;
– fournir une boîte à outils pour la modélisation d'IP. Pour effectuer de nouvelles démonstrations de sécurité d'IP, on peut en effet non seulement employer la méthodologie que nous avons mise en œuvre mais on peut également envisager de partager une partie des développements Coq en les rendants plus génériques : comparaison des termes par interprétation polynomiale et ordre lexicographique comme nous l'avons fait, normalisation par récriture, ...

Alors que les techniques actuelles de sécurisation d'une interface de programmation reposent plutôt sur des méthodes heuristiques de conception et de gestion des problèmes en aval — détection des attaques et mécanismes d'alerte, maîtrise des conséquences financières des attaques, refus (éventuellement malhonnête) des banques de reconnaître les détournements pour ne pas mettre en difficulté l'ensemble du système bancaire — on peut espérer que Coq, sans avoir besoin de chanter, sera plus efficace contre certains attaquants que les oies du Capitole.

Références

[Bon01] Mike Bond. Attacks on cryptoprocessor transaction sets. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 220–234. Springer, 2001.

[Bon04] Mike Bond. *Understanding Security APIs*. PhD thesis, University of Cambridge Computer Laboratory, June 2004.

[Ott] Otter : An automated deduction system. Site web : http ://www-unix.mcs.anl.gov/AR/otter/.

[Wik] SSL Acceleration. http ://en.wikipedia.org/wiki/SSL_acceleration.

[YAB+05] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, Computer Laboratory, August 2005.