Louis Goubin – Professeur – Laboratoire PRiSM – Université de Versailles-St-Quentin-en-Yvelines – Louis.Goubin@uvsq.fr
Olivier Ly – Maître de conférences – LaBRI – Université de Bordeaux – ly@labri.fr

SÛRETÉ DE FONCTIONNEMENT ET SÉCURITÉ DES ALGORITHMES CRYPTOGRAPHIQUES

mots-clés : *cryptographie / attaques physiques / méthodes formelles*

Dans la conception traditionnelle de la cryptologie, la sécurité est vue de manière abstraite : pour attaquer un cryptosystème, l'attaquant se borne à échanger des messages avec celui-ci, et espère en les utilisant pouvoir mettre en défaut les objectifs visés (confidentialité, intégrité, authenticité,…) par différentes techniques de cryptanalyse. La cryptographie classique s'efforce donc de construire des schémas avec si possible des preuves relatives de sécurité contre ce type d'attaque, en admettant la difficulté de certains problèmes mathématiques au sens de la théorie de la complexité.

1. Introduction

Depuis une dizaine d'années, l'étude de la sécurité des solutions à base de cartes à puce a montré qu'il est nécessaire de tenir compte également des attaques physiques. Ce nouveau concept prend en considération non seulement la sécurité des cryptosystèmes au sens mathématique, mais aussi les aspects liés à la nature des calculs. Ces attaques sont particulièrement menaçantes pour les systèmes embarqués, comme peuvent l'être les cartes à microprocesseur, contre lesquels l'adversaire peut mobiliser des moyens d'analyse de plus en plus sophistiqués.

En particulier, une des hypothèses implicites qui est faite dans le modèle de sécurité abstrait est que le fonctionnement de la carte à puce est exactement conforme aux spécifications, qu'il s'agisse de la couche matérielle (processeur, mémoires, bus de communication) ou de la couche logicielle.

Or, dans la pratique, on s'est aperçu que cette « sûreté de fonctionnement » n'est pas toujours garantie. En particulier, plusieurs défauts peuvent apparaître. Tout d'abord, des erreurs de calcul (dysfonctionnement du microprocesseur) peuvent être provoquées intentionnellement par un attaquant. Ce genre d'attaque « active » fait partie des attaques physiques les plus simples et les moins coûteuses à mener. Ces attaques constituent d'ailleurs une menace non seulement pour les algorithmes cryptographiques, mais aussi pour d'autres composantes logicielles, comme la machine virtuelle Java [1] ou, plus globalement, le système d'exploitation dans son ensemble [2].

Par ailleurs, même si aucune attaque active n'est utilisée, un autre type de dysfonctionnement peut « simplement » résulter d'un « bug », que ce soit dans la couche logicielle ou la couche matérielle d'un carte à puce. Nous illustrerons cela par un scénario récemment proposé par Adi Shamir pour l'algorithme RSA.

Par conséquent, s'assurer que le fonctionnement d'une carte à puce est conforme aux spécifications (matérielles et logicielles) n'est pas seulement important pour garantir la disponibilité du service, mais aussi pour se prémunir contre des attaques dévastatrices. Sûreté de fonctionnement et sécurité sont en fait intimement liées. Le développement de méthodes de plus en plus sophistiquées pour garantir le fonctionnement correct des logiciels embarqués est donc de plus en plus crucial, et nous verrons que, dans cette optique, les méthodes formelles jouent un rôle de plus en plus important.

2. Principe des attaques par injection de fautes

Le fait pour un attaquant de provoquer intentionnellement des erreurs de fonctionnement dans un dispositif électronique faisant des calculs fait partie de la catégorie des attaques dites « actives ». Dans le cas d'une carte à puce, il s'agit de modifier l'environnement physique de la carte pour la placer dans des conditions anormales de fonctionnement.

Plusieurs moyens sont à la disposition de l'attaquant.

- L'alimentation électrique : selon le standard ISO/IEC 7816-2, le micro-module doit pouvoir supporter une tension d'alimentation Vcc comprise entre 4,25 et 5,25 volts. Pour ces valeurs, la carte doit fonctionner normalement. En revanche, si une variation brusque de l'alimentation (appelée « *spike* ») fait sortir Vcc de l'intervalle de tolérance, cela peut provoquer un résultat faux, à supposer que la carte soit capable d'achever le calcul.
- L'horloge : de façon analogue, le standard ISO/IEC 7816-2 définit une fréquence de référence pour l'horloge externe, ainsi qu'un intervalle de tolérance. L'utilisation d'une fréquence anormalement haute ou basse peut également provoquer des erreurs.
- La température : placer la carte dans des conditions de températures extrêmes est un moyen potentiel de provoquer des fautes, même s'il est assez peu utilisé aujourd'hui dans la pratique.
- Les rayonnements : le folklore présente souvent les attaques par injection de faute comme les « attaques au micro-ondes » (l'attaquant plaçant la carte à puce dans un four à micro-ondes pour lui faire calculer des résultats erronés). Au-delà de cette vision un peu caricaturale, il est reconnu que des rayonnements correctement dirigés peuvent influencer le comportement de la carte.
- La lumière : l'illumination d'un transistor peut le faire basculer temporairement dans son état conducteur, provoquant ainsi une erreur. En appliquant une source de lumière intense (produite par une lampe flash d'appareil photographique, amplifiée par un microscope), on peut changer la valeur de bits individuels dans une mémoire SRAM. Par la même technique, on peut également interférer avec les instructions, perturbant ainsi des sauts conditionnels.
- Les courants de Foucault : induits par un champ magnétique dans une bobine, ils peuvent par exemple provoquer des erreurs dans une cellule de mémoire (qu'elle soit de type RAM, EPROM, EEPROM ou Flash).

3. Exemple d'attaque par faute sur l'algorithme RSA

En ce qui concerne les cryptosystèmes, c'est en septembre 1996 que trois chercheurs de Bellcore, Dan Boneh, Richard DeMillo et Richard Lipton, proposent un nouveau modèle d'attaque physique sur les cartes à microprocesseur, qu'ils baptisent « *cryptanalysis in the presence of hardware faults* ». Ce modèle d'attaque est alors dirigé contre plusieurs algorithmes cryptographiques à clé publique : le système RSA et les schémas d'authentification de Fiat-Shamir et de Schnorr.

...Illustrons l'attaque dans le cas où l'algorithme RSA est implémenté en utilisant la technique des « restes chinois »...

Mathématiquement, on peut décrire l'algorithme RSA de la manière suivante. On commence par choisir l'exposant public e (des exemples courants sont $e=3$, $e=17$, $e=257$ ou $e=65537$). On utilise ensuite un générateur de nombres aléatoires pour obtenir deux nombres premiers p et q, tels que e soit premier avec $p-1$ et avec $q-1$. Si on pose $n=p\times q$, la clé publique est alors constituée de e et de n, alors que la clé secrète (ou privée) est constituée de p et q. Typiquement, on prend souvent actuellement p et q de taille 512 bits,

et donc n de taille 1024 bits. La fonction de chiffrement est alors définie par $f: x \rightarrow y = x^e \ mod \ n$ et la fonction de déchiffrement par $f^{-1}: y \rightarrow x = y^d \ mod \ n$, où d est l'inverse de e modulo $ppcm(p-1, q-1)$. Ici d est également une valeur qui doit rester secrète.

Illustrons l'attaque dans le cas où l'algorithme RSA est implémenté en utilisant la technique des « restes chinois » (qui présente l'intérêt d'accélérer les calculs d'un facteur environ 4 par rapport à une implémentation classique). Pour le calcul de $x = y^d \ mod \ n$, l'idée est de calculer séparément $x_p = x \ mod \ p$ et $x_q = x \ mod \ q$, puis de reconstituer x à partir de x_p et x_q en utilisant le théorème des restes chinois. Pour cela, on suppose que sont stockées dans la carte à puce les valeurs $d_p = d \ mod \ (p-1)$ et $d_q = d \ mod \ (q-1)$. Les valeurs x_p et x_q sont calculées au moyen des formules suivantes : $x_p = y_p^{dp} \ mod \ p$ et $x_q = y_q^{dq} \ mod \ q$, où $y_p = y \ mod \ p$ et $y_q = y \ mod \ q$.

L'attaquant lance alors le calcul deux fois. La première fois sans introduire de perturbation. Il obtient alors les bonnes valeurs pour x_p et x_q et le théorème des restes chinois donne la valeur correcte x. La seconde fois, il perturbe le calcul (par exemple au moyen de rayonnements électromagnétiques ou en faisant varier l'alimentation électrique...). Il obtient par exemple la bonne valeur pour x_p, mais une valeur erronée (disons x'_q) pour x_q. Le théorème des restes chinois donne alors une valeur x' erronée, qui vérifie $x' \equiv x \ mod \ p$, mais $x' \neq x \ mod \ q$. Cela montre que $x' - x$ est divisible par p mais pas par q. On peut alors calculer $PGCD(x' - x, n)$, qui est alors égal à p. La factorisation de n est donc trouvée, et le système est cassé !

Notons qu'un moyen simple d'éviter ce type d'attaque consiste à vérifier systématiquement le calcul avant de sortir le résultat. Pour le RSA, c'est particulièrement commode, puisqu'il suffit par exemple de s'assurer que $x^e = y \ mod \ n$.

4. Attaques par bug

Dans les attaques par injection de faute décrites ci-dessus, nous avons supposé qu'un attaquant perturbe en temps réel le fonctionnement du microprocesseur pendant l'exécution des calculs. Dans un autre scénario récemment décrit par Adi Shamir et al, baptisé « *bug attack* » [5], on fait l'hypothèse qu'il existe un « bug », soit dans la couche logicielle, soit dans la couche matérielle d'un dispositif effectuant des calculs cryptographiques.

Dans l'attaque par faute habituelle, il est nécessaire d'avoir accès physiquement au dispositif de calcul pendant l'exécution du calcul, ce qui rend l'attaque réaliste pour des systèmes embarqués comme les cartes à puce, mais beaucoup moins quand il s'agit d'un serveur distant. Au contraire, la seule présence d'un bug peut être suffisante pour monter une attaque dévastatrice, y compris sur un serveur distant (via internet par exemple).

...Un bug de la multiplication entière peut résulter en une attaque, à nouveau sur le cryptosystème RSA....

Pourquoi s'intéresse-t-on de plus en plus à un tel scénario ? Cela vient principalement de deux facteurs : l'accroissement des tailles des registres et l'optimisation de plus en plus fine des opérations arithmétiques dans les microprocesseurs modernes. Par conséquent, la probabilité qu'un bug se cache dans ces opérations devient de moins en moins négligeable. À titre d'exemple, on peut citer la découverte accidentelle d'un bug dans l'opération de division sur le Pentium en 1994 (couche matérielle), et la découverte d'un bug dans la multiplication pour le logiciel Excel en 2007 (couche logicielle).

Illustrons comment un bug de la multiplication entière peut résulter en une attaque, à nouveau sur le cryptosystème RSA.

L'hypothèse de départ est que le microprocesseur manipule des « mots » (typiquement de taille 32 ou 64 bits), et qu'il existe deux mots a et b tels que le produit $a \times b$ soit calculé de manière incorrecte. On suppose en outre que l'attaquant connaît ces deux valeurs a et b, soit parce qu'il a découvert le bug accidentellement, soit parce qu'il est complice avec un des concepteurs du microprocesseur qui aurait introduit volontairement le bug.

On suppose par ailleurs que l'algorithme RSA (utilisé en mode signature ou en mode déchiffrement) est implémenté en utilisant la technique des « restes chinois ». Le dispositif visé prend en entrée une valeur y, et renvoie le résultat $x = y^d \ mod \ n$, qu'il obtient en calculant séparément $x_p = y_p^{dp} \ mod \ p$ et $x_q = y_q^{dq} \ mod \ q$, où $y_p = y \ mod \ p$, $y_q = y \ mod \ q$, $d_p = d \ mod \ (p-1)$ et $d_q = d \ mod \ (q-1)$, puis en reconstituant x à partir de x_p et x_q en utilisant le théorème des restes chinois.

L'idée principale de l'attaque est la suivante. On peut toujours supposer, quitte à changer les notations, que $p<q$. Puis, on choisit comme valeur d'entrée un entier c tel que $p<c<q$. Il suffit pour cela de prendre pour c la partie entière de $\sqrt{n}$. Ensuite, on modifie légèrement c, en remplaçant ses deux mots de poids le plus faible respectivement par a et b. L'entier y obtenu est la valeur que l'on fournit en entrée à l'algorithme RSA. Il est facile de voir qu'avec une grand probabilité, cette valeur y vérifie également $p<y<q$.

Lors du calcul de RSA, les grands entiers sont divisés en « mots » (typiquement de 32 ou 64 bits). La multiplication de

deux grands entiers se fait par la méthode classique, qui fait intervenir tous les produits possibles entre un mot du premier entier et un mot du second entier.

La méthode des restes chinois sépare le calcul RSA en deux phases :

- Le calcul modulo q commence par calculer $y_q = y \bmod q$, c'est-à-dire $y_q = y$, puisque y<q par hypothèse. Pour calculer l'exponentielle $x_q = y_q^{d_q} \bmod q = y^{d_q} \bmod q$, on effectue des multiplications successives, dont en particulier y^2. Le fait que les deux mots de poids faible de y valent respectivement a et b provoque alors le « bug » lors du calcul de $y^2 = y \times y$. Le résultat est donc erroné pour x_q.
- En revanche, comme y>p, la réduction $y_p = y \bmod p$ donne une valeur y_p avec une probabilité négligeable d'avoir ses deux mots de poids faible égaux à a et b. Par conséquent, le « bug » a une probabilité négligeable de se produire, et le résultat obtenu pour $y_p^{d_p} \bmod p$ est très certainement correct.

Le résultat final obtenu est donc une valeur x' erronée (au lieu de la valeur correcte x), qui vérifie $x' \equiv x \bmod p$, mais $x' \neq x \bmod q$. Cela implique en particulier que $x'^e = y \bmod p$, mais $x'^e \neq y \bmod q$. Cela montre que $x'^e - y$ est divisible par p mais pas par q. On peut alors calculer $PGCD(x'^e - y, n)$, qui est alors égal à p. La factorisation de n est donc trouvée, et le système est cassé !

Même si, dans la pratique, cette attaque n'a pas (encore) eu de conséquence directe, elle est une illustration frappante du risque que des bugs peuvent faire peser sur la sécurité des algorithmes cryptographiques. Ce problème est encore accru par l'augmentation de la taille des registres dans les microprocesseurs les plus modernes (64, voire 128 bits) qui rend infaisable la vérification du bon fonctionnement d'un multiplieur par un test exhaustif. Par ailleurs, on a donné ici un exemple de bug « matériel », mais l'existence potentielle de bugs « logiciels » fait peser le même type de risque sur la sécurité. Toutes ces raisons imposent de mettre en place une méthodologie adaptée lors du développement des microprocesseurs et des logiciels pour garantir leur conformité aux spécifications, en particulier lorsqu'il s'agit d'applications « sensibles ».

5. Méthodes formelles

Les bugs et autres erreurs de conception font partie de la vie du programmeur. Quiconque ayant écrit ne serait-ce que quelques lignes de code le sait bien. Développer un logiciel sans bug est difficile. Appliquer soigneusement les méthodes de génie logiciel classiques ou moins classiques comme l'« *extreme programming* » améliore les choses sans aucun doute. Et c'est même absolument nécessaire dès lors que le projet atteint une taille critique. Comment un projet faisant intervenir plusieurs dizaines de personnes, ou plusieurs équipes, pourrait arriver à terme sans une méthode claire de développement ? Celle-ci permettra d'avoir une vue globale, d'intégrer les différents travaux, et de chasser les bugs de façon systématique à tous les niveaux au moyen de batteries de tests sophistiquées.

...Certains secteurs d'activités ne peuvent accepter l'existence de bugs : Les transports, l'avionique ou encore le secteur bancaire ont besoin d'une confiance absolue en leurs logiciels : ce sont les « logiciels critiques »...

Mais, il n'en reste pas moins que cela ne donne pas l'assurance absolue que le logiciel produit est exempt de bug. Par exemple, les systèmes d'exploitation les plus répandus, pourtant développés dans les règles de l'art avec des moyens colossaux, comportent des bugs et c'est d'ailleurs uniformément accepté. De fait, dans la plupart des cas, l'industrie du logiciel accepte de mettre sur le marché des logiciels bogués. Ce qui semble raisonnable puisque dans la plupart de ces cas, en effet, les bugs en question ne prêtent pas vraiment à conséquence. Et l'utilisateur préfère bénéficier de nouvelles fonctionnalités même s'il doit le payer par quelques bugs. On peut observer ce phénomène à l'extrême dans le secteur des téléphones mobiles dont les durées de vie sont très courtes et les temps de développement raccourcis au-delà du minimum vital.

À l'opposé, certains secteurs d'activités ne peuvent accepter cela. Les transports, l'avionique ou encore le secteur bancaire ont besoin d'une confiance absolue en leurs logiciels : ce sont les « logiciels critiques ». Il serait par exemple inacceptable d'admettre le déploiement de logiciels non sûrs dans un avion ou un train. Il en va de même pour le logiciel gérant votre compte en banque. Dans ce type de domaine, le déploiement d'un logiciel nécessite d'avoir un degré de confiance maximum.

En science, se convaincre d'un fait donné utilise essentiellement deux voies : l'expérimentation et la démonstration, au sens mathématique du terme. Pour ce qui concerne le logiciel, le test correspond à l'expérimentation. Les méthodes formelles explorent quant à elles la voie de la démonstration.

Conservons ici le terme « explorer », car il s'agit d'un domaine en plein développement du point de vue de la recherche scientifique.

Les tests, aussi sophistiqués soient-ils, ne peuvent que très rarement énumérer tous les cas. Considérons simplement une fonction écrite en langage C comportant 5 paramètres de type int ; tester tous les cas d'entrée possible coûterait 2^{4x8x5}, soit 2^{160} exécutions de la fonction, ce qui est hors de question. Bien sûr, l'analyse du code de la fonction permet en général d'éliminer la plupart des cas et de concentrer les tests sur certains cas critiques. Pour la plupart des cas d'application, c'est tout à fait satisfaisant. Mais, ce faisant, on ajoute tout de même une incertitude. Ajoutons que les batteries de tests sont elles-mêmes développées par des humains, et sont également sujettes aux erreurs.

...Les méthodes formelles visent à donner au logiciel le même degré de confiance qu'un théorème mathématique en produisant des démonstrations....

Les méthodes formelles visent à donner au logiciel le même degré de confiance qu'un théorème mathématique en produisant des démonstrations. Pour comprendre de quoi il retourne, illustrons cela par un exemple. On sait qu'il existe une infinité de nombres premiers. Mais finalement comment sait-on cela ? Une première méthode pour s'en assurer pourrait consister à utiliser un ordinateur pour énumérer le plus possible de nombres premiers. Les ordinateurs modernes sont puissants, il est certain que l'on pourrait en énumérer énormément. Serait-ce satisfaisant ? Pas du point de vue mathématique. On dirait plutôt par exemple que s'il n'en existait qu'en nombre fini, on pourrait les noter p_0, p_1, p_2, p_3,... , p_n. On pourrait alors considérer le nombre $p = 1 + p_0 \times p_1 \times p_2, p_3 \times \ldots \times p_n$. Ce nombre ne serait alors divisible par aucun des p_k. Sinon, le nombre 1 le serait également, ce qui est faux. Par conséquent p serait premier. Or, il n'était pas listé dans les p_k. Ce qui est contradictoire. Il y a donc une infinité de nombres premiers. Ce raisonnement est une démonstration. Il nous donne l'assurance maximale que le fait est correct.

Que cela donne-t-il pour un programme ? Considérons par exemple une fonction de tri :

```
1:      void tri(int t[], int n) {
2:       int k, chgt = 1;
3:        while (chgt)
4:         for (k=0, chgt=0; k<n-1; k++)
5:            if (t[k] > t[k+1]) {
6:                echange(t,k,k+1); // échange les contenus des
                  cases d'indices k et k+1 dans le tableau t
7 :               chgt = 1;
8 :            }
9:      }
```

Ce tri est extrêmement mauvais du point de vue des performances. Il peut bien sûr être amélioré en bien des points, mais il va nous faciliter la présentation. Le code source de la fonction `echange(int t[], int i, int j)` est omis. Il consiste simplement à échanger les contenus des cellules d'indices `i` et `j` dans le tableau `t`.

Cette fonction prend donc en entrée un tableau d'entiers `t`, et sa taille `n`. Elle trie le tableau. Comment démontre-t-on cela ? On considère les couples d'indices `(i,j)` du tableau tels que `i<j` et `t[i]>t[j]`. Appelons-les « inversions ». On peut observer facilement que le tableau `t` est trié si et seulement si il ne comporte pas d'inversion. Observons également deux faits : D'abord, l'échange de la ligne 6, lorsqu'il est exécuté, fait toujours décroître strictement le nombre global d'inversions dans le tableau. En effet, l'échange en élimine une et on peut vérifier facilement qu'il n'en ajoute aucune. Ensuite, s'il existe des inversions dans le tableau, alors il en existe au moins une qui est contiguë (une inversion de type `(i,i+1)`). En effet, supposons qu'il existe au moins une inversion ; choisissons-en une, disons `(i,j)`, pas nécessairement contiguë, mais telle que `j` soit minimal. Dans ce cas, `(i,j-1)` n'est pas une inversion, sinon `j` n'a pas été bien choisi. Et donc `t[i] <= t[j-1]`. Or `t[i] > t[j]`. Donc `t[j-1]>t[j]`, ce qui prouve que `(j-1,j)` est une inversion contiguë.

De cela, on déduit que lorsque le programme entre dans la boucle à la ligne 4, si le tableau comporte une inversion, alors le nombre global d'inversion diminuera strictement après la boucle. En effet, nous avons vu que l'existence d'une inversion implique celle d'une inversion contiguë. Si nous choisissons l'inversion contiguë d'indice minimum, alors elle sera corrigée par l'échange à coup sûr lorsque la boucle interne y parviendra. Si, au contraire, le tableau ne comporte aucune inversion à l'entrée de la boucle de la ligne 4, alors, en particulier, il n'y en a aucune de contiguë, et donc la variable `chgt` restera à 0 et le programme se terminera. Cela veut dire en particulier que le tableau est trié.

Ainsi, si le tableau comporte un nombre `N` d'inversions au début de l'exécution du programme, alors il y aura au plus `N` tours de boucle, puisque chaque tour fait diminuer strictement le nombre d'inversion. Le programme s'arrêtera donc au bout d'un temps fini, et le tableau sera trié. C'est ce que nous voulions. Nous avons établi la démonstration que cette fonction trie bien le tableau.

Cela dit, les choses ne sont pas si simples. Par exemple, nous avons omis certaines hypothèses nécessaires à la bonne exécution de la fonction. Il faut que `n` soit positif sans quoi la boucle de la ligne 4 pourrait ne pas terminer ; il faut également que le tableau `t` soit alloué en mémoire, sans quoi les accès

risquent de provoquer des erreurs. Une démonstration complète nécessite la prise en compte de tous les détails, d'autant plus que le plus souvent, les bugs se cachent dans les détails.

Sans être forcément difficiles, les démonstrations de programme sont souvent complexes et laborieuses. Les méthodes formelles consistent à élaborer et à vérifier ce type de démonstrations à l'aide de logiciels : les logiciels d'aide à la démonstration (méthode B, Coq, PVS, etc.). L'ordinateur pourra traiter les cas faciles et vérifier une foule de détails, l'utilisateur devra donner les idées clés de la démonstration.

Le principe est de voir une démonstration comme un enchaînement d'applications de règles de raisonnement logique. Ces règles peuvent être codées dans un ordinateur qui pourra dès lors vérifier des démonstrations avec la plus grande rigueur et de façon exhaustive.

Par ailleurs, la démonstration de la validité d'un programme nécessite de formaliser sa spécification dans le langage des mathématiques. Par exemple, pour exprimer qu'un programme trie les n premiers éléments d'un tableau t, il faudra écrire $\forall\ i, j \in [0,n-1] : i \leq j \Rightarrow t[i] \leq t[j]$. C'est certes une contrainte, mais cela oblige le programmeur ou le concepteur à rédiger ses spécifications avec la plus grande rigueur, ce qui en soi améliore le processus de développement. Notons que cette tâche est souvent difficile en elle-même et fait parfois appel à des concepts sophistiqués. C'est le cas lorsqu'en sécurité on veut formaliser l'idée de confidentialité par exemple.

On sait qu'on ne pourra jamais élaborer de logiciel qui produira les démonstrations de façon complètement automatisée. C'est une conséquence simple de l'indécidabilité du problème de l'arrêt des machines de Turing : il n'existe pas de programme qui décide si un autre programme s'arrête ou non. Par exemple, votre compilateur préféré n'a pas d'option pour dire si le programme qu'il compile s'arrête ou non. Cela veut dire que les logiciels d'aide à la démonstration nécessiteront toujours une intervention de l'utilisateur. Par exemple, si nous revenons à l'exemple des nombres premiers, l'ordinateur est capable de produire et de vérifier toute la démonstration, sauf l'idée de choisir p, qui devra être suggéré par l'utilisateur. En revanche, si l'on restreint les cas d'étude ou bien le type de propriété que l'on veut vérifier, on peut parfois obtenir des programmes qui décident de façon totalement autonome la véracité de telle ou telle propriété, c'est le « *model checking* ». C'est en particulier le cas si à la place de programmes, on cherche à vérifier les propriétés de machines à état-transition finies.

Conclusion

Les méthodes formelles font toujours aujourd'hui l'objet de recherches académiques très actives. Néanmoins, il faut noter déjà un certain nombre de succès industriels. On peut citer le projet METEOR pour la ligne 14 du métro parisien. Le logiciel embarqué a été développé au moyen de la méthode B. On peut également citer la certification critères communs EAL7 produite par la société Gemalto qui a nécessité la modélisation et la vérification formelle en Coq d'une carte à puce embarquant une machine virtuelle Java. Aujourd'hui, les critères de certification de logiciel, pour les niveaux les plus élevés, intègrent en effet l'utilisation des méthodes formelles comme une nécessité. ■

Bibliographie

[1] GOVINDAVAJHALA (S.), APPEL (A.W.), « *Using Memory Errors to Attack a Virtual Machine* », *IEEE Symposium on Security and Privacy*, Oakland, 2003. Disponible sur **www.cs.princeton.edu/~sudhakar/papers/memerr.pdf**.

[2] AKKAR (M.-L.), GOUBIN (L.), LY (O.), « *About an Automatic Fault Injection Protection System* ». In *Proceedings of E-Smart'2003*, Nice, 2003.

[3] BONEH (D.), DEMILLO (R.A.), LIPTON (R.J.), « *On the Importance of Checking Cryptographic Protocols for Faults* », In *Proceedings of EUROCRYPT'97*, LNCS 1233, pp. 37-51, Springer-Verlag, 1997.

[4] BERZATI (A.), CANOVAS (C.), GOUBIN (L.), « *Perturbating RSA Public Keys: An Improved Attack* », In *Proceedings of CHES 2008*, LNCS 5154, pp. 380-395, Springer-Verlag, 2008.

[5] BIHAM (E.), CARMELI (Y.), SHAMIR (A.), « *Bug Attacks* », In *Proceedings of CRYPTO 2008*, LNCS 5157, pp. 221-240, Springer-Verlag, 2008.

[6] ANDRONICK (J.), CHETALI (B.), LY (O.), « *Using Coq to Verify Java Card Applet Isolation Properties* », In *proceedings of 16th Theorem Proving in Higher Order Logic*, Roma (TPHOL'2003), LNCS 2758, pp. 335-351, Springer-Verlag, 2003.