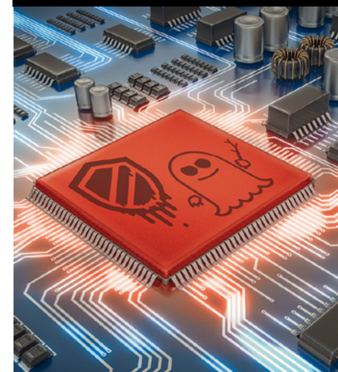


VÉRIFIER UN CODE PIN

Ronan LASHERMES – ronan.lashermes@inria.fr
SED & LHS / INRIA-RBA
Hélène LE BOUDER – helene.le-bouder@imt-atlantique.fr
IMT-Atlantique

mots-clés : CODE PIN / ATTAQUES PHYSIQUES / AUTHENTIFICATION / CARTES À PUCE / SYSTÈMES EMBARQUÉS

Entrer son code PIN pour utiliser sa carte bancaire ou déverrouiller son téléphone portable est devenu un geste quotidien. L'objet doit vérifier que le code proposé est correct. Comment implémenter cette vérification ? Cela semble être une simple comparaison de deux tableaux de données. Détrompez-vous ! Les attaques physiques vont nous mener la vie dure.

L'utilisation d'un code PIN (*Personnal Identification Number*) est une méthode répandue pour authentifier un utilisateur. Seul l'utilisateur légitime connaît le code PIN correct, celui-ci étant généralement un nombre à quatre chiffres. À un moment se pose nécessairement le problème de vérifier si le code PIN proposé par l'utilisateur, appelé le PIN candidat, est correct ou non. Nous allons voir que cette vérification n'est pas évidente à mettre en œuvre **[thèse Rivière]**.

Nous considérons ici que la cible est un microcontrôleur : pas de caches complexes, un seul cœur, une vitesse d'horloge réduite, les instructions sont exécutées dans l'ordre... Dans cet article, oubliez Spectre et Meltdown, nous retournons aux bases des attaques physiques, où l'attaquant interagit physiquement avec le circuit cible.

1 Votre mission si vous l'acceptez...

1.1 Le problème

Attention !

Comme pour la cryptographie, une règle d'or pour la vérification de code PIN est : ne l'implémentez jamais vous-même ! Il y a plus d'attaques à prendre en compte que celles présentées dans cet article d'introduction.

Votre patron vous a demandé de réaliser une application (en C) pour un système embarqué (téléphone portable, carte bancaire, objet connecté...) qui doit authentifier l'utilisateur. Pas de panique, vous connaissez l'existence des codes PIN. L'utilisateur doit mémoriser un code à quatre chiffres préalablement généré aléatoirement par le système : le `pin_correct`, stocké en mémoire non-volatile.

Évidemment pour se protéger des attaques par force brute, c'est-à-dire une attaque qui consiste à tester toutes les 10000 possibilités, un nombre d'essais maximal est autorisé. Si l'utilisateur se trompe 3 fois de suite, le système doit considérer qu'il s'agit d'une attaque et réagir pour se protéger. Le circuit peut être bloqué temporairement, voire se « suicider ». L'idée c'est que le circuit ne soit plus utilisable par l'attaquant : il s'autodétruit. Par exemple, il peut y avoir effacement des secrets (clés cryptographiques, `pin_correct`...).

Si le code PIN proposé par l'utilisateur (`pin_candidat`) est correct, la méthode `authenticate()` est appelée pour prendre en compte l'authentification réussie. Si le code candidat est faux (différent du `pin_correct`), mais qu'il reste des essais, la méthode `incorrect()` est appelée pour signaler l'erreur et le compteur d'essais `try_counter` est décrémenté. S'il ne reste plus d'essais, l'autodestruction est obtenue par l'appel à la méthode `kill()`.

1.2 Première implémentation naïve

La vérification de code PIN s'insère normalement dans un protocole d'authentification complet, souvent complexe. Ici nous nous intéressons uniquement à la phase de vérification du secret.

Voici une première implémentation. Pour minimiser la quantité de code ici, nous allons omettre de contrôler les domaines de définition des variables. Petit défi : essayez d'identifier un maximum de vulnérabilités dans ce code.

Article publié sous licence (cc) BY-NC-ND

```
// V1
#define PIN_SIZE 4
#define MAX_TRY 3
typedef unsigned char BYTE;
int try_counter = MAX_TRY ;
//...
bool compare_arrays(BYTE *arr1, BYTE *arr2) { // équivalent à memcmp
  for(int i = 0 ; i < PIN_SIZE ; i++) {
    if(arr1[i]!=arr2[i]) return false ;
  }
  return true ;
}

void verify_pin(BYTE *pin_candidat, BYTE *pin_correct) {
  if(compare_arrays(pin_candidat, pin_correct) == true) { // PIN bon
    try_counter = MAX_TRY ; // réinitialise try_counter au max
    authenticate() ;
  }
  else { // PIN incorrect
    try_counter -= 1 ; // 1 essai de moins
    if(try_counter <= 0) {
      kill() ; // 3, 2, 1, ...  BOUM
    }
    else {
      incorrect() ; // le prochain essai est peut être le bon ?
    }
  }
}
```

Et là, les ennuis commencent...

1.3 Votre adversaire

L'objet (carte à puce, téléphone...) responsable de la vérification du code PIN peut être volé. Nous supposons donc qu'il faut faire face à un attaquant qui a un accès physique au circuit. Il peut tester le circuit à sa guise dans son laboratoire de super-vilain.

Quelle que soit la robustesse théorique de l'algorithme utilisé, celui-ci peut être la cible d'une attaque physique. En effet, les attaques physiques n'attaquent pas l'algorithme en lui-même, mais son implémentation logicielle et matérielle. Elles sont divisées en deux familles : les attaques par observation de paramètres physiques et les attaques par injections de faute.

Une attaque par observation analyse les caractéristiques physiques d'un circuit afin de retrouver des données protégées. Car si l'algorithme utilisé est abstrait, le circuit lui est bien concret. Il a donc un temps de calcul, une consommation de courant, un rayonnement électromagnétique, une température, etc. On parle de fuite physique d'information par un canal auxiliaire.

Une attaque par injection de fautes consiste à altérer le comportement normal d'un circuit cible en modifiant ses conditions normales de fonctionnement. Il existe de nombreuses méthodes pour injecter une faute. Lorsqu'un attaquant avance un front montant de l'horloge, on parle de perturbation d'horloge. Une perturbation d'alimentation consiste à sous-alimenter une puce durant un temps très court. Des techniques plus avancées sont les injections électromagnétiques et les injections laser. Plus précisément une impulsion électromagnétique envoyée localement sur un circuit peut générer une faute. De même, après ouverture du boîtier autour de la cible (le silicium doit être exposé), l'attaquant qui illumine des transistors cibles à l'aide d'un laser focalisé génère des courants créant des fautes.

L'implémentation de la vérification de code PIN doit prendre en compte toutes ces menaces : le secret ne doit jamais sortir de l'objet responsable de la vérification.

2 Le temps joue contre vous

2.1 Attaque par mesure de temps et par arrachage

La première fuite physique qui a été utilisée contre des codes PIN était le temps de calcul. En effet, si un temps d'exécution dépend du secret, il y a une fuite d'information à exploiter.

Ici, l'attaquant est capable d'évaluer le temps d'exécution de `compare_arrays` en mesurant le temps de réponse du circuit. Ainsi, il teste toutes les possibilités pour le premier chiffre, tout en mesurant le temps d'exécution. Par exemple, 0 donne 12,1 µs, 1 donne 12,2 µs, 2 donne 14.3 µs... Comme le temps de calcul pour 2 est le plus long, l'attaquant sait maintenant qu'il s'agit du premier chiffre. Il peut ainsi recommencer pour le deuxième chiffre, puis pour le troisième et le quatrième. Le problème ici est que la méthode de comparaison du PIN candidat avec celui correct n'est pas en temps constant. La mesure du temps d'exécution fait donc fuir de l'information permettant de retrouver le code PIN.

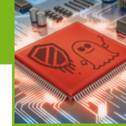
Le lecteur astucieux remarque alors, que pour retrouver le code PIN complet en utilisant la mesure du temps, l'attaquant a besoin de plus de 3 essais... Très bonne remarque ! L'attaquant exploite en fait une deuxième vulnérabilité : la possibilité d'arracher l'objet à sa vérification. Le terme arrachage vient de l'acte de retirer violemment une carte à puce de son lecteur dans le but d'interrompre l'exécution du programme en cours. Plus généralement, « l'arrachage » peut être obtenu sur tout système dont on peut rapidement interrompre l'alimentation électrique. Ainsi si le temps d'exécution pour `compare_arrays` est court, l'alimentation est coupée rapidement. L'objectif est que la ligne `try_counter -= 1 ;` ne soit jamais exécutée, permettant ainsi un nombre d'essais illimité. En combinant cette technique avec la mesure de temps, l'attaquant est en mesure de retrouver le code PIN secret.

2.2 La parade

Il va donc falloir se protéger de cette attaque :

- avec une comparaison en temps constant ;
- en s'assurant que l'arrachage ne donne pas d'essais supplémentaires ;

Article publié sous licence (cc) BY-NC-ND



- en s'assurant que le compteur d'essais soit écrit en mémoire non volatile au bon endroit.

```
// V2
//...
bool compare_arrays(BYTE *arr1, BYTE *arr2) { // équivalent à
memcmp
  BOOL result = true ;
  int i ;
  for(i = 0 ; i < PIN_SIZE ; i++) {
    result = result && (arr1[i]==arr2[i]) ? true : false ;
  }
  return result ;
}

void verify_pin(BYTE *pin_candidat, BYTE *pin_correct) {
  try_counter = readNVM(try_counter_address) - 1;
  writeNVM(try_counter_address, try_counter) ;
  if(compare_arrays(pin_candidat, pin_correct) == true) {
    // PIN bon
    writeNVM(try_counter_address, MAX_TRY) ;
    authenticate() ;
  }
  else { // PIN incorrect
    if(try_counter <= 0) {
      kill() ; // 3, 2, 1, ...  BOUM
    }
    else {
      incorrect() ; // le prochain essai est peut être le bon ?
    }
  }
}
```

3 Une toute petite faute de rien du tout

Ça y est, votre implémentation est en temps constant. L'attaquant ne peut plus récupérer le bon code PIN en mesurant le temps d'exécution. On devrait être bon, non ? Hélas, non !

3.1 Saut d'instruction

Il est vrai que l'attaquant peut avoir du mal à observer l'exécution du programme pour retrouver le secret. Mais il peut aussi attaquer par injection de faute. Son but est alors de forcer la puce à ne pas vérifier le PIN candidat.

Pour cela, il positionne à proximité immédiate du circuit une antenne en champ proche. Il fait ensuite circuler un courant très fort durant un temps très court (les paramètres exacts dépendent de la puce ciblée). Par couplage électromagnétique (similaire aux plaques à induction en cuisine), une partie de cette énergie va être transmise dans les lignes métalliques à la surface de la puce (pistes d'alimentation, d'horloge, bus de données...). Un courant est généré dans la puce, en conséquence des erreurs peuvent apparaître. Une faute matérielle peut avoir divers effets sur un programme : inversion de bit(s), modification de valeur, saut d'instruction. C'est ce dernier modèle de faute qui nous intéresse ici : l'attaquant est capable de sauter une instruction du programme cible.

Le code assembleur obtenu (extrait, compilé pour ARMv7-M) ressemble à ceci :

```
bl compare_arrays // appelle compare_arrays
mov r3, r0 //r0 contient le résultat de compare_arrays
cmp r3, #0 // les tableaux sont-ils identiques ?
beq pin_incorrect //sinon PIN incorrect
...
bl authenticate
```

Il suffit à l'attaquant de supprimer la bonne instruction (**`beq pin_incorrect`**) avec une injection de faute pour forcer l'appel à **`authenticate()`**. Dans ce cas, le branchement vers les instructions considérant le PIN candidat incorrect n'est jamais exécuté : la vérification du PIN candidat n'est même pas prise en compte.

3.2 La parade

Comment se protège-t-on d'un attaquant ayant le pouvoir de supprimer dynamiquement une instruction ? Comme cette action est dynamique (en cours d'exécution du programme), cela ne sert à rien de vérifier le programme avant de l'exécuter, il est tout à fait correct. En fait, se protéger contre ce genre d'attaque est un sujet de recherche actif.

Il y a plusieurs stratégies possibles. La plus simple est la redondance. Si les opérations les plus critiques sont répétées, il y a moins de chance que l'attaquant réussisse à fauter les deux opérations au cours de la même exécution. Il faut réécrire le programme tel que la suppression d'une instruction n'ait pas d'effet sur la fonctionnalité.

```
// V3
  adr lr, fin_compare // on charge explicitement
  adr lr, fin_compare // l'adresse de retour de la fonction
  b compare_arrays
  b compare_arrays
fin_compare :
  mov r3, r0 //r0 contient le résultat de compare_arrays
  cmp r3, #0 // les tableaux sont-ils identiques ?
  beq pin_incorrect //sinon PIN incorrect
  mov r3, r0 //r0 contient le résultat de compare_arrays
  cmp r3, #0 // les tableaux sont-ils identiques ?
  beq pin_incorrect //sinon PIN incorrect
  ...
  bl authenticate
```

Ce genre de programme redondant est assez difficile à écrire à la main. Ici par exemple le résultat de la comparaison est mémorisé sur **`r0`** uniquement : il n'y a pas de redondance à ce niveau et une faute opportune dans **`compare_arrays`** peut mettre à mal notre protection (si cette fonction n'est pas proprement protégée).

On doit également se méfier du compilateur qui est parfois capable d'optimiser du code redondant : il faut toujours s'assurer que la redondance est bien présente dans le binaire, pas uniquement dans le code source. D'autre part, *inliner* une fonction permet de se protéger d'un attaquant capable de sauter l'appel à cette fonction.

Article publié sous licence CC BY-NC-ND

La redondance peut même être automatisée à la compilation pour doubler les instructions par exemple **[Compilation]**. Mais elle n'est pas très efficace. Oui, elle ralentit un peu l'attaquant, mais il est possible expérimentalement de réaliser plusieurs fautes au cours de la même exécution **[Multi-fautes]**. De plus, le coût (en temps de calcul, d'énergie...) de cette protection est important.

Une autre stratégie est l'utilisation de contre-mesures matérielles : il est possible de détecter des tentatives d'injections de faute en plaçant de nombreux petits détecteurs à la surface de la puce. Ici la difficulté est de distinguer une attaque (avec la volonté de compromettre le circuit) d'une simple radiation. Il serait dommage que votre carte de paiement devienne inutilisable à cause d'un simple voyage en avion !

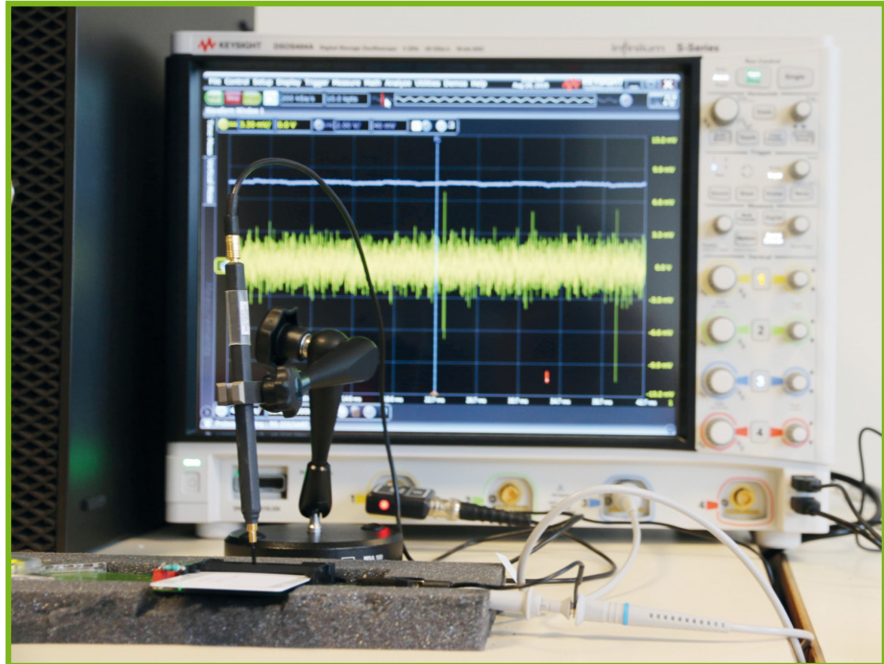


Fig. 1 : Une expérience d'écoute électromagnétique au LHS à l'INRIA de Rennes.

Une autre contre-mesure matérielle est l'utilisation d'un bouclier (*shield*) : de nombreuses lignes métalliques sont placées à la surface de la puce. Des données pseudo-aléatoires sont écrites d'un côté, et lues à l'autre bout de la ligne : si ces données sont erronées, c'est que soit une attaque les a modifiées, soit la ligne a été coupée pour accéder au silicium (ce qui représente également une attaque). Un bouclier bien réalisé est très performant et permet même de générer du bruit électromagnétique contre les attaques par observation.

Enfin, il existe des techniques de protection par vérification de l'intégrité du flot de contrôle, dont nous avons déjà parlé dans l'épisode précédent **[MISC96/Fautes]**.

4 Ça fuit toujours !

La première attaque présente une fuite d'information par le temps de calcul, cependant il existe d'autres fuites physiques comme la consommation de courant et les émissions électromagnétiques. Les codes PIN ont longtemps été épargnés par les attaques qui exploitent de telles fuites, car elles nécessitent souvent l'acquisition de données pour de nombreuses exécutions. Or dans une vérification de code PIN l'attaquant n'a que 3 essais et donc que 2 exécutions à analyser pour retrouver le secret avant le blocage. Cependant, une attaque par caractérisation (apprentissage supervisé) est possible **[Template Attack]**.

On suppose que l'attaquant possède un circuit identique à celui qu'il veut attaquer ; dont il est l'utilisateur légitime. Il y a donc deux circuits :

- un qui est la cible et où il n'a que 3 essais pour retrouver le code PIN ;
- un circuit de référence sur lequel l'attaquant connaît le PIN et peut le changer selon son souhait (cela reste une supposition forte).

La première étape de l'attaque est de caractériser le circuit de référence par rapport à une fuite d'information. Les mesures obtenues sont en fait une « signature » de l'exécution du programme au sein du circuit (instructions exécutées, données manipulées...). Sans être capable d'identifier exactement une instruction ou des données manipulées, le biais statistique présent dans un signal peut être suffisant pour une exploitation. Après caractérisation, selon nos résultats expérimentaux, il est possible de distinguer l'exécution d'une instruction à la place d'une autre. Il est même possible de retrouver la valeur d'une donnée chargée dans un registre avec une probabilité non négligeable. Beaucoup de mesures doivent être obtenues, le nombre dépendant du bruit présent dans le signal. La caractérisation permet d'apprendre le signal significatif et les caractéristiques du bruit.

Ainsi, l'attaquant acquiert des mesures de la consommation de courant ou d'émissions électromagnétiques lors de la vérification de code PIN en faisant varier le PIN candidat et le vrai PIN. Il s'agit de son circuit, il peut donc changer le PIN correct à sa guise et acquérir des mesures de vérification de PIN candidat incorrect en alternant avec un PIN candidat correct et ainsi ne jamais bloquer son circuit de référence. L'attaquant collecte des mesures de PIN correct 0000, puis 1111, ..., 9999 tout en faisant varier les valeurs du PIN candidat (là encore 0000, 1111, ...). À la fin, il obtient des mesures pour les 100 couples de chiffres (candidat, correct). En découpant ses mesures en 4, et en les recombinant l'attaquant a de quoi reconstituer toutes les comparaisons de codes PIN possible. Généralement, les attaques par observation sont du type diviser pour régner. Ce qui signifie que l'on attaque le secret (ici le PIN code) par

Article publié sous licence (cc) BY-NC-ND

petits morceaux. Ici, les 4 chiffres sont attaqués en parallèle indépendamment les uns des autres.

Si la première étape d'apprentissage est un peu longue, la phase d'attaque, elle, est très rapide. L'attaquant réalise une première mesure sur le circuit cible. Il propose un PIN candidat au hasard. La mesure obtenue est confrontée aux résultats de la caractérisation (à l'aide d'une distance de Mahalanobis entre notre mesure et les 100 caractérisations) et donne deux types de résultats :

- elle valide avec quasi-certitude un chiffre correct ;
- dans le cas contraire, elle suggère un chiffre plus probable.

Par exemple, si le vrai PIN est 1234 et que l'attaquant propose 0256, son outil de caractérisation va lui valider le deuxième chiffre : il s'agit bien du 2. En effet, quelle que soit la méthode choisie pour comparer les valeurs, il y a une différence de comportement dans le programme face à une égalité ou à une inégalité. Aussi, la caractérisation permet de détecter en une courbe cette différence. Pour les autres chiffres la caractérisation peut suggérer à l'attaquant quels chiffres ont une forte chance d'être bons. Cela signifie que le circuit fuit directement de l'information sur la valeur des chiffres. Ainsi au deuxième essai, si la caractérisation est performante, l'attaquant sait qu'il peut proposer 1234. Pour une caractérisation un peu moins précise, il a besoin de plus d'essais pour réussir son authentification.

Si pour vous protéger des attaques en fautes vous avez doublé la comparaison de code PIN, votre caractérisation est plus précise et lors de l'attaque, vous avez deux mesures correspondant à la comparaison au lieu d'une. Ainsi la contre-mesure de la partie précédente introduit une vulnérabilité supplémentaire face à ce type d'attaque. Cette attaque ne permet pas toujours de retrouver le code PIN en 3 essais, mais la probabilité d'y arriver est élevée. Elle peut être déployée par un attaquant qui cherche à casser de nombreux objets identiques à partir d'une unique phase de caractérisation.

Pour se protéger de ce type d'attaques, il est primordial de réduire le rapport signal sur bruit des fuites d'information. Une protection possible est l'utilisation de générateurs de bruit matériels pour réduire la qualité des mesures de l'attaquant. D'autre part, on cherche à masquer les données critiques : au lieu de manipuler les données directement, on manipule une valeur modifiée par un masque aléatoire. À la fin du calcul, on démasque les données pour retrouver le bon résultat. Ce démasquage est possible si les bonnes opérations sont effectuées sur la valeur du masque seul en parallèle du calcul principal.

Petit exemple pour un cas simple, nous voulons masquer une opération linéaire : `y = f(x)` (la linéarité implique que pour tout `x` et `x'`, `f(x+x') = f(x) + f(x')`). Dans ce cas, nous pouvons utiliser une valeur aléatoire `m` pour masquer `x` et calculer `y = f(x+m)–f(m)`. Ainsi, `x` n'est pas manipulé directement et ne peut donc pas fuir. Pour retrouver `x`, l'attaquant doit combiner 2 fuites : une pour retrouver `x+m` et une autre pour retrouver `m`. Ce type d'attaque est dit d'ordre 2, car sa complexité a augmenté.

Article publié sous licence (cc) BY-NC-ND

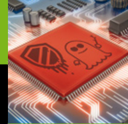
ACTUELLEMENT DISPONIBLE !

LINUX PRATIQUE n°107

PISTAGE SUR LE WEB : COMMENT S'EN PRÉMUNIR ?

NE LE MANQUEZ PAS CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

https://www.ed-diamond.com



5 Pourquoi faire simple quand on peut faire compliqué ?

Est-il impossible de simplement comparer deux tableaux et agir en conséquence de manière sûre ? En plus de mettre en place les contre-mesures venant d'être présentées (calcul en temps constant, de la consommation de courant, contre les fautes...), il est possible de jouer sur le protocole.

En effet, l'objet auquel nous voulons nous authentifier a-t-il besoin de connaître la valeur correcte du PIN ? Non, de la même manière qu'un serveur web ne doit jamais mémoriser les mots de passe de ses utilisateurs, mais uniquement l'image de ce mot de passe auquel on a appliqué une fonction de hachage cryptographique.

Une fonction de hachage calcule une empreinte de taille fixe appelé haché à partir d'une entrée qui peut être de taille quelconque. Lorsqu'il s'agit en plus d'une fonction de hachage cryptographique, cela signifie que la fonction respecte certaines propriétés de sécurité : l'impossibilité de retrouver la valeur d'origine à partir du haché et une résistance aux collisions.

Une première hypothèse est simplement de mémoriser le haché du code PIN sur l'objet. Toutefois, il n'y a que 10000 codes PIN différents et donc que 10000 hachés différents. Si on récupère le haché (avec une attaque physique), il est trivial de retrouver le code PIN original en pré-calculant tous les hachés possibles. Dans notre cas, nous utilisons donc un HMAC, c'est-à-dire la combinaison d'une fonction de hachage cryptographique avec une clé secrète : la connaissance de cette clé est nécessaire pour obtenir le résultat du hachage.

Lors de l'initialisation, une clé secrète `K` est générée en même temps que `pin_correct` et les valeurs secrètes `K` et `hash_correct = HMAC(K, pin_correct)` sont sauvegardées sur l'objet. Il est normalement impossible, sans connaître la clé secrète, de remonter à `pin_correct`.

Lors de la vérification, l'objet calcule `hash_candidat = HMAC(K, pin_candidat)`. La comparaison n'a donc pas lieu sur les valeurs des codes PIN, mais sur les valeurs des hachés (`hash_correct` et `hash_candidat`). Ainsi, une attaque en temps ou par caractérisation peut permettre de récupérer `hash_correct`, mais sans la clé `K`, cela n'est d'aucune utilité. Une attaque par caractérisation n'est plus possible, car la clé secrète diffère entre l'objet cible et l'objet de référence servant à l'apprentissage. Par contre, une attaque par injections de faute peut toujours se révéler efficace.

En utilisant ce protocole, faut-il toujours sécuriser sa comparaison ? Cela n'est pas indispensable, car on ne peut remonter au secret sans la clé `K`. C'est d'ailleurs le choix retenu par Android qui utilise un simple `memcmp_s` pour la comparaison des hachés (`memcmp_s` n'est pas en temps constant). Mais d'un autre côté, une vérification en temps constant des hachés serait peu coûteuse et ne pas laisser fuiter `hash_correct` permet de ne pas aider l'attaquant. Qui sait si une autre attaque ne permet pas de récupérer `K` ?

Note

Ce bout de protocole s'insère normalement dans un protocole d'authentification plus complexe. Il s'agit ici d'un exemple simplifié, il ne faut pas l'utiliser tel quel en pratique. Malheureusement, il est difficile de trouver une implémentation sécurisée d'une vérification de code PIN dans la nature. Il s'agit d'un secret précieusement gardé par les industriels concernés. Une exception notable est l'implémentation faite par Android, dans le sous-système gatekeeper.

Conclusion

Dans le contexte d'un circuit sujet aux attaques physiques, implémenter une comparaison sécurisée relève du parcours du combattant. Le choix de nos outils n'a pas aidé, peut-être que le langage C n'est pas le plus adapté pour une implémentation sécurisée. En pratique, on préfère utiliser un langage plus contraignant qui permet de limiter les possibilités d'erreurs et on utilise des outils de vérification statique et de preuves formelles (par exemple Frama-C).

Les techniques montrées dans cet article peuvent être utilisées pour d'autres cas de comparaisons sécurisées. Par exemple dans le cadre d'un Secure Boot, il faut s'assurer que l'image à charger n'a pas été modifiée, par exemple en comparant le haché de l'image à la signature sauvegardée en mémoire.

Finalement, même si le développeur peut prendre quelques mesures pour se prémunir de certaines attaques, les meilleures solutions sont un mélange de modifications matérielles de la puce (boucliers, détecteurs d'injection, générateurs de bruit...) et du choix du bon protocole par rapport à la fonctionnalité demandée. ■

Remerciements

Merci à Jean-Louis Lanet et Laurent Clévy pour leurs relectures.

Références

[Template Attack] H. Le Bouder et al., « A template attack against VERIFY PIN algorithms », SECRYPT 2016

[Multi-fautes] L. Rivière, Thèse : « Sécurité des implémentations logicielles face aux attaques par injection de faute sur systèmes embarqués », 2015

[Compilation] T. Barry et al, « Compilation of a Countermeasure Against Instruction-Skip Fault Attacks », CS2@HiPEAC 2016

[MISC96/Fautes] R. Lashermes, « Attaques par faute », *MISC n°96*, mars-avril 2018

[thèse Rivière] L. Rivière et al., « High precision fault injections on the instruction cache of ARMv7-M architectures », HOST 2015

Article publié sous licence (cc) BY-NC-ND