Microsoft Research - Inria
JOINT CENTRE

Composants logiciels vérifiés en F* : Poly1305

Jean-Karim ZINZINDOHOUÉ
INRIA Paris

Benjamin BEURDOUCHE
INRIA Paris & MSR-INRIA

EQUIPE PROJET
PROSECCO
CENTRE PARIS

1 JUIN 2016

SOMMAIRE

1

La sécurité des composants critiques en question

Pourquoi utiliser des méthodes formelles ?

OpenSSL: Version C de Poly1305 (avril 2016)

```
201         /* last reduction step: */
202         /* a) h2:h0 = h2<<128 + d1<<64 + d0 */
203         h0 = (u64)d0;
204         h1 = (u64)(d1 += d0 >> 64);
205         h2 += (u64)(d1 >> 64);
206         /* b) (h2:h0 += (h2:h0>>130) * 5) %= 2^130 */
207         c = (h2 >> 2) + (h2 & ~3UL);
208         h2 &= 3;
209         h0 += c;
210         h1 += (c = CONSTANT_TIME_CARRY(h0,c));   /* doesn't overflow */
```

Pourquoi utiliser des méthodes formelles ?

OpenSSL: Version C de Poly1305 (avril 2016)

```
201         /* last reduction step: */
202         /* a) h2:h0 = h2<<128 + d1<<64 + d0 */
203         h0 = (u64)d0;
204         h1 = (u64)(d1 += d0 >> 64);
205         h2 += (u64)(d1 >> 64);
206         /* b) (h2:h0 += (h2:h0>>130) * 5) %= 2^130 */
207         c = (h2 >> 2) + (h2 & ~3UL);
208         h2 &= 3;
209         h0 += c;
210         h1 += (c = CONSTANT_TIME_CARRY(h0,c));    /* doesn't overflow */
```

It does

Pourquoi utiliser des méthodes formelles ?

- Tests non exhaustifs
  - Exemple d'OpenSSL: occurrence du dépassement mémoire trop rare ($< 2^{-64}$) pour être détectée
- HeartBleed (2014) : la sûreté mémoire est cruciale
- Protection contre les attaques par canaux auxiliaires
  - "ECDH Key-Extraction via Low-Bandwidth Electromagnetic Attacks on PCs" par Genkin, Pachmanov, Pipman & Tromer

Un langage (très) fortement typé

F* en quelques points (www.fstar-lang.org) :

- Langage open-source « à la ML »
  - fonctionnel, non orienté objet
- Système de types riche
  - Raffinements logiques, types dépendants, modèle mémoire etc.
- Interaction avec un solver SMT
  - Automatisation des preuves
- Effacement vers OCaml/F#

Garanties obtenues via F*

Propriétés prouvées statiquement, par typage :

- Correction fonctionnelle du code
- Sûreté mémoire et prévention des dépassements d'entiers
- Discipline de programmation spécifique : ni branchements ni accès mémoires dépendants de valeurs sensibles

2

Cas concret : Poly1305

Un algorithme de MAC

L'algorithme Poly1305

- Algorithme de « codes d'authentification de messages » (MAC)
- Moderne (RFC mai 2015), proposé par D. Bernstein
- Basé sur de l'arithmétique modulo P = $2^{130} - 5$

L'algorithme Poly1305

```
poly1305_mac(msg, key):
  r = key[0..15] & 0x0ffffffc0ffffffc0ffffffc0fffffff
  s = key[16..31]
  a = 0
  p = (1<<130)-5
  for i=1 upto ceil(length msg/ 16)
    n = msg[((i-1)*16)..(i*16)] | [0x01]
    a += n
    a = (r * a) % p
    end
  a += s
  return to_bytes(a)
  end
```

L'algorithme Poly1305

```
poly1305_mac(msg, key):
  r = key[0..15] & 0x0ffffffc0ffffffc0ffffffc0fffffff
  s = key[16..31]
  a = 0
  p = (1<<130)-5
  for i=1 upto ceil(length msg/ 16)
    n = msg[((i-1)*16)..(i*16)] | [0x01]
    a += n
    a = (r * a) % p
    end
  a += s
  return to_bytes(a)
  end
```

Dépassement dans OpenSSL

Un type spécifique aux valeurs sensibles

Élaboration d'un type spécifique pour les entiers machine :

```
new type sint (* Type abstrait pour les valeurs sensibles *)

val v: sint -> GTot int (* Valeur mathématique des entiers *)

type usint (n:nat) = x:sint{v x >= 0 /\ v x < 2^n}
                     (* Un entier machine de 'n' bits *)
```

Chemins d'exécution indépendants des secrets

Discipline de programmation :

- Secrets représentés par le type *abstrait* **sint**
- Pas de comparaison possible (hormis via masquage)
  - Pas de branchements conditionnels dépendants de secrets
  - Pas d'accès mémoire dépendants de secrets

Résultat : Le chemin d'exécution est indépendant des valeurs sensibles dans le code source

Sûreté mémoire et dépassements

Spécifications contraintes des opérateurs sur le type `sint` :

```
val zero: z:usint 64{v z = 0} (* Constante 0x00ul *)

val add64: a:usint 64 -> b:usint 64 ->
        Tot (c:usint 64{v a + v b < 2^64 ==> v c = v a + v b})
        (* Addition pour 2 entiers machines de 64 bits *)
```

Correction fonctionnelle

Retour sur l'algorithme :

```
poly1305_mac(msg, key):
  r = key[0..15] & …
  s = key[16..31]
  a = 0
  p = (1<<130)-5
  for i=1 upto ceil(length msg/ 16)
    n = …
    a += n
    a = (r * a) % p
  …
```

Correction fonctionnelle

```
a += n
a = (r * a) % p
```

Dépassement dans OpenSSL

Correction fonctionnelle

```
val add_and_multiply: a:bigint -> n:bigint{…} -> r:bigint{…} ->
   ST unit
   (requires (fun h -> …))
   (ensures (fun h0 _ h1 -> … /\
    eval h1 a % p = ((eval h0 a + eval h0 n) * eval h0 r) % p))
```

Spécifications F*

```
a += n
a = (r * a) % p
```

Dépassement dans OpenSSL

Correction fonctionnelle

```
val add_and_multiply: a:bigint -> n:bigint{…} -> r:bigint{…} ->
   ST unit
   (requires (fun h -> …))
   (ensures (fun h0 _ h1 -> … /\
    eval h1 a % p = ((eval h0 a + eval h0 n) * eval h0 r) % p))
```

Etat mémoire avant appel à la fonction

Spécifications F*

```
a += n
a = (r * a) % p
```

Dépassement dans OpenSSL

Correction fonctionnelle

```
val add_and_multiply: a:bigint -> n:bigint{…} -> r:bigint{…} ->
   ST unit
   (requires (fun h -> …))
   (ensures (fun h0 _ h1 -> … /\
    eval h1 a % p = ((eval h0 a + eval h0 n) * eval h0 r) % p))
```

Etat mémoire AVANT appel à la fonction

Etat mémoire APRES appel à la fonction

Spécifications F*

```
a += n
a = (r * a) % p
```

Dépassement dans OpenSSL

Correction fonctionnelle

```
val add_and_multiply: a:bigint -> n:bigint{…} -> r:bigint{…} ->
   ST unit
   (requires (fun h -> …))
   (ensures (fun h0 _ h1 -> … /\
   eval h1 a % p = ((eval h0 a + eval h0 n) * eval h0 r) % p))
```

Etat mémoire AVANT appel à la fonction

Etat mémoire APRES appel à la fonction

Spécifications F*

Fonction reliant un tableau d'entiers à sa valeur mathématique

```
a += n
a = (r * a) % p
```

Dépassement dans OpenSSL

3

Perspectives

Perspectives

F* évolue et progresse en permanence :

- Amélioration du support : mode interactif, extraction vers OCaml et F#
- Nouveaux compilateurs en cours de développement : C, C++, Javascript
- Application à des projets d'envergure : miTLS*

Merci

www.fstar-lang.org

poly1035 code :

https://github.com/FStarLang/FStar/tree/master/examples/primitives

SSTIC 2016
RENNES

www.inria.fr