HACL⋆, une bibliothèque cryptographique formellement vérifiée dans Firefox

Benjamin Beurdouche[1] et Jean-Karim Zinzindohoué[2]
benjamin.beurdouche@inria.fr
jean-karim.zinzindohoue@interieur.gouv.fr

[1] INRIA Paris
[2] Ministère de l'intérieur

Résumé. Les protocoles de communications sécurisés tels que Transport Layer Security (TLS) fondent leur sécurité sur un ensemble d'algorithmes cryptographiques. Ces primitives sont des composants logiciels critiques dans le fonctionnement et la sécurité des protocoles et se doivent donc d'être implémentées de manière robuste. Il est malheureusement fréquent que ces primitives contiennent des erreurs d'implémentation mettant en péril cette sécurité. Dans cet article nous décrivons HACL⋆ (High Assurance Crypto Library) une bibliothèque cryptographique écrite en F⋆ contenant un ensemble d'algorithmes nécessaires aux opérations cryptographiques les plus répandues. Nous décrivons ensuite l'ensemble du processus ayant permis sa mise en production au sein de la bibliothèque cryptographique Network Security Services (NSS), utilisée entre autres par Firefox[3], le navigateur web de Mozilla. Pour finir nous proposons un retour d'expérience sur l'emploi des méthodes formelles et la mise en place de techniques à l'état de l'art dans un produit utilisé quotidiennement par plus de cent millions d'utilisateurs.

1 Introduction

Les protocoles de communication sécurisés tels que Transport Layer Security (TLS) fondent leur sécurité sur un ensemble d'algorithmes cryptographiques. Ces primitives sont des composants logiciels critiques dans le fonctionnement et la sécurité des protocoles et se doivent donc d'être implémentées de manière robuste. Il est malheureusement fréquent que ces primitives contiennent des erreurs d'implémentation mettant en péril l'ensemble de la sécurité du protocole ou de l'application. Dans la suite nous décrivons HACL⋆ (High Assurance Crypto Library), une bibliothèque cryptographique formellement vérifiée contenant un ensemble d'algorithmes nécessaires aux opérations cryptographiques les plus répandues. Nous

[3] https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/

décrivons ensuite l'ensemble du processus ayant permis sa mise en production au sein de la bibliothèque cryptographique Network Security Services (NSS), utilisée entre autres par Firefox, le navigateur web de Mozilla, ainsi que le retour d'expérience lié a l'emploi des méthodes formelles et des techniques à l'état de l'art dans un produit utilisé quotidiennement par plus de cent millions d'utilisateurs. Nous souhaitons par là illustrer que le niveau de maturité actuel des méthodes formelles est suffisant pour leur intégration dans des processus industriels et encourager académiques et industriels à poursuivre leur coopération dans ce domaine. L'ensemble du code est publiquement accessible[4] sous une licence MIT.

2 HACL⋆, une bibliothèque cryptographique formellement vérifiée

2.1 De nouvelles implémentations de primitives cryptographiques pour parer au manque de confiance

Implémenter une primitive cryptographique a ceci de difficile qu'il faut à la fois qu'elle soit sûre — c'est à dire qu'il n'y ait ni problèmes de sûreté mémoire, ni erreurs fonctionnelles, ni de canaux cachés — et que le code soit le plus performant possible. En effet, prenons par exemple le cas d'un échange de clés entre un client et un serveur. Si le client ne se rendrait pas forcément compte d'une différence de temps de calcul de quelques centièmes de secondes entre deux implémentations, le serveur, qui initie un grand nombre de connexions, aura besoin que tout l'échange soit optimisé. Toute inefficacité induirait en effet un surcoût pour le fournisseur du service, que ce soit en termes de ressources matérielles, de consommation d'énergie ou en latence. Cela devient encore plus critique lorsqu'un tunnel sécurisé est mis en place. L'algorithme de chiffrement utilisé constitue alors un goulot d'étranglement, susceptible de limiter les performances de l'ensemble de la communication. Or ces deux objectifs, sécurité et performance, s'opposent. Chaque nouvelle optimisation apportée fait peser le risque d'introduire une nouvelle vulnérabilité. Les méthodes traditionnelles de tests unitaires sont peu utiles étant donné que les espaces des valeurs prises en entrée sont gigantesques. Par conséquent, la probabilité de rencontrer certains bugs lors de tests sur des valeurs aléatoires est négligeable. Cela n'exclut malheureusement pas un attaquant averti de tirer parti de ces bugs. Il s'agit bien là d'une réalité, comme en attestent les nombreuses failles de sécurité corrigées chaque année[5] dans la bibliothèque OpenSSL [9],

[4] `https://github.com/mitls/hacl-star`
[5] `https://www.openssl.org/news/vulnerabilities.html`

pourtant l'une des plus attentivement revues par la communauté — au moins depuis la publication de la faille Heartbleed [7] en 2014.

2.2 Les vertus des méthodes formelles

Puisqu'il est difficile d'exclure totalement la présence d'un bug dans une primitive cryptographique, que ce soit à l'aide de tests unitaires ou de techniques de *fuzzing*, les méthodes formelles proposent une approche différente. En effet, il s'agit de démontrer de façon statique, c'est à dire une fois pour toutes à la compilation du code source, qu'un fragment de code vérifie certaines propriétés. Celles-ci reposent généralement sur une notion de *contrat* : si certaines hypothèses sont vraies pour les arguments d'une fonction avant l'execution de cette fonction (les *préconditions*), alors l'outil prouve formellement que d'autres propriétés sont vraies après l'execution de celle-ci (les *postconditions*). À titre d'exemple simple, on pourra montrer que la somme de deux entiers positifs (précondition) est un entier positif (postcondition). Il est bien évidemment possible d'aller beaucoup plus loin dans la finesse et la complexité des propriétés démontrées. C'est l'objectif de HACL*, une bibliothèque cryptographique dont chacune des primitives a été formellement vérifiée. Notons cependant que les méthodes formelles n'ont pas toujours bonne presse en dehors du monde académique, pour plusieurs raisons :

- le langage source est peu connu et/ou difficile à appréhender et/ou mal maintenu ;
- le langage source est bien connu (par exemple du langage C), mais doit se conformer à une syntaxe et une structure inhabituelle et restrictive pour être exploitable par les outils de preuve formelle ;
- le langage cible, s'il y en a un, est difficilement lisible (code généré par une machine) ;
- les propriétés formellement vérifiées sont obscures et/ou mal comprises de la communauté ;
- les performances du produit sont insuffisantes.

Le reste de cette section présente HACL*, sa structure et la méthodologie employée pour répondre à chacune des difficultés énoncées ci-dessus.

2.3 Formellement vérifiée, certes, mais que prouve-t-on ?

Notre méthodologie s'appuie sur le langage F* [2,10,11] et son compilateur vers le langage C, KreMLin [5]. F* est un langage fonctionnel orienté vers

la vérification. Il dispose d'un système de types très riche lui permettant de rajouter des raffinements logiques aux objets manipulés. Par exemple, il est possible de spécifier qu'une variable n'a pas le type entier int, mais le type des entiers naturels, plus précis (car restreint aux entiers positifs ou nuls. Pour ce faire, une annotation logique est placée entre accolades après le type F⋆ : n:int{n ≥ 0}.

Le système de vérification semi-automatisé s'appuie sur un prouveur de théorème (*SMT solver*), en l'occurrence Z3 [6], pour s'assurer statiquement de la correction des assertions logiques dans le code. Afin de permettre une compilation efficace vers C, un fragment du langage F⋆ que nous appelons Low⋆ (pour *low F⋆*) a été isolé, et une preuve de la correction de la compilation de Low⋆ vers C a été présentée dans de précédents travaux [5]. Cette preuve garantit que les propriétés énoncées et prouvées au niveau du code source (en F⋆) sont bien conservées par la compilation vers C.

Afin de simplifier le processus de compilation (et la preuve de correction de celle-ci), certaines propriétés complexes du langage C n'ont pas de correspondance en Low⋆. Par exemple, il n'est pas possible d'y faire référence à l'adresse mémoire d'une variable (opérateur & en C) ou encore de réaliser des conversions entre entiers et pointeurs. De ce fait, l'image de Low⋆ après compilation est seulement un fragment du C. Ce fragment se situe dans la sémantique opérationelle de CompCert [8], un compilateur certifié écrit en Coq. Les propriétés énoncées et prouvées dans le code source écrit en Low⋆ sont donc préservées par la compilation vers C, puis potentiellement vers le code machine en utilisant CompCert.

Le langage F⋆ n'étant pas des plus répandus, nous nous sommes attachés à cloisonner les différentes parties du code au sein de la bibliothèque HACL⋆, afin d'accroître sa lisibilité. Nous souhaitons qu'un utilisateur de la bibliothèque puisse facilement retrouver et comprendre le fonctionnement de celle-ci, ainsi que les garanties offertes. Ainsi la bibliothèque se découpe-t-elle en trois parties :

1. les spécifications ;
2. le code optimisé en Low⋆;
3. le code généré en C.

Les spécifications Les spécifications sont la référence *fonctionnelle* des primitives cryptographique de HACL⋆ : chaque primitive extraite en C a le même comportement fonctionnel que sa spécification en F⋆ correspondante, c'est à dire qu'elles ont le même comportement d'entrées/sorties. Les

spécifications et le code optimisé correspondant utilisent les mêmes types de données pour les entrées et les sorties (typiquement des chaînes d'octets). La preuve de correction fonctionnelle d'HACL⋆ garantit statiquement que, quelles que soient les valeurs d'entrée, si celles-ci respectent la précondition de la primitive, alors la spécification et le code optimisé produisent des résultats identiques.

Il est donc essentiel que ces spécifications soient revues par un panel de contributeurs aussi large que possible afin de s'assurer de leur correction. Au demeurant, lors du développement de la bibliothèque, plusieurs garde-fous ont déjà été mis en place afin d'accroître le niveau de confiance dans les spécifications fonctionnelles :

1. ces spécifications sont extrêmement proches de leur pendant textuel (RFC ou spécification FIPS du NIST) ;
2. elles sont écrites dans un fragment pur du langage F⋆ et sont très succinctes (moins de 100 lignes de code par primitive) ;
3. elles ont déjà fait l'objet d'une revue attentive ;
4. elles sont exécutables, et passent les tests unitaires.

En effet, afin de permettre au plus grand nombre de les relire, ces spécifications sont écrites dans un fragment pur de F⋆ et sont aisément lisibles par quiconque est un peu familier avec la programmation fonctionnelle. À titre d'exemple, le listing de la figure 1 montre comment sont spécifiées les opérations dans le corps $\mathbb{Z}/(2^{255} - 19)\mathbb{Z}$.

```
1 (* Field types and parameters *)
2 let prime = pow2 255 − 19
3 type elem : Type0 = e:int{e ≥ 0 ∧ e < prime}
4 let fadd e1 e2 = (e1 + e2) % prime
5 let fsub e1 e2 = (e1 − e2) % prime
6 let fmul e1 e2 = (e1 ∗ e2) % prime
7 let zero : elem = 0
8 let one : elem = 1
```

Fig. 1. Spécification du corps $\mathbb{Z}/(2^{255} - 19)\mathbb{Z}$ en F⋆

Le code Low⋆ (et le code C) correspondant à ces opérations est lui optimisé, il utilise des tableaux d'entiers, et met en œuvre des mesures de protections contre les canaux auxiliaires etc., il est donc bien plus évident de lire cette spécification. Parce qu'elles sont très proches du document de référence (RPC ou FIPS) qui les décrit, il est également aisé d'en faire la revue en s'appuyant sur la description et le pseudo code fournis par

ces publications. Enfin, F⋆ étant un véritable langage de programmation, ce code, bien que peu efficace, peut être exécuté grâce au processus de compilation standard de F⋆. Le code est alors d'abord compilé vers OCaml, puis d'OCaml vers un exécutable. Ces spécifications sont ainsi soumises à un ensemble de tests unitaires pour plus de sûreté.

Le code Low⋆ optimisé Si les spécifications sont concises et faciles à lire, ce n'est pas le cas du code Low⋆. Celui-ci nécessite pour être relu un certain niveau d'expertise dans le système de preuve de F⋆ et dans la primitive cryptographique elle-même, afin de bien comprendre comment la preuve de correction est menée. En revanche la lecture de l'API de HACL⋆ permet de bien comprendre les propriétés prouvées.

Celles-ci se déclinent en trois parties : sûreté mémoire, correction fonctionnelle et exécution indépendante des secrets.

La sûreté mémoire est garantie par le système de type de Low⋆ lui-même. Une publication précédente [5] présente les mécanismes en détails, mais l'intuition est la suivante : contrairement à la spécification scalarmult (voir figure 2), la fonction crypto_scalarmult (voir figure 3) n'est pas *pure*. En effet, elle interagit avec la mémoire du programme. Cette interaction se manifeste à travers l'annotation Stack, qui signifie que toutes les allocations sont faites sur la pile et que rien n'a été alloué ou désalloué sur le tas. Par ailleurs le système garantit qu'un pointeur ne peut être déréférencé que s'il pointe vers une région mémoire valide (live). C'est la raison pour laquelle cette fonction vient avec un contrat dont la précondition est une fonction de h, la mémoire du programme. Celui-ci garantit que les arguments de la fonction (output, secret et point, qui sont chacun des tampons de 32 octets) ont bien déjà été alloués en mémoire. Cette hypothèse est explicitée par le prédicat live, où live mem buf signifie que le tampon buf est bien alloué dans la mémoire mem.

D'autre part, la postcondition prend, elle, en argument la mémoire initiale h0, le résultat retourné res (inutilisé ici) et la mémoire h1 du programme quand la fonction retourne. Elle fournit ici deux garanties importantes. La première est que seule le tampon output a été modifié, c'est ce qu'indique le prédicat modifies_1 output h0 h1.

La correction fonctionnelle est la seconde garantie apportée par la post-condition. En effet, h1.[output] == Spec.scalarmult h0.[secret] h0.[point] indique que la valeur de output dans l'état final est égale à l'application de la spécification scalarmult sur les valeurs de secret et de point dans l'état initial.

Cette propriété est démontrée quelle que soit la valeur de secret et de point en entrée de la fonction, pourvu évidemment que ces deux tampons soient bien alloués. La fonction est ainsi prouvée correcte, sous réserve que la spécification elle-même le soit, comme évoqué plus haut.

```
let lbytes (l:nat) = b:seq FStar.UInt8.t{length b = l}

let scalarmult (k:lbytes 32) (u:lbytes 32) : Tot (lbytes 32) =
  let k = decodeScalar25519 k in
  let u = decodePoint u in
  let res = montgomery_ladder u k in
  encodePoint res
```

Fig. 2. Spécification de la fonction d'exponentiation de Curve25519

```
type uint8_p = buffer Hacl.UInt8.t

val crypto_scalarmult:
  output:uint8_p{length output = 32} →
  secret:uint8_p{length secret = 32} →
  point:uint8_p{length point = 32} →
  Stack unit
    (requires (λ h → live h output ∧ live h secret ∧ live h point))
    (ensures (λ h0 res h1 → live h1 output ∧ modifies_1 output h0 h1 ∧
     live h0 output ∧ live h0 secret ∧ live h0 point ∧
     h1.[output] == Spec.scalarmult h0.[secret] h0.[point]))
```

Fig. 3. Signature Low⋆ de la fonction d'exponentiation de Curve25519

L'indépendance des traces d'exécution vis-à-vis des secrets est la dernière propriété illustrée par cette API. Les octets vers lesquels pointent les tampons output, secret et point ont un type particulier dans la bibliothèque HACL⋆, qui les distingue des valeurs publiques. Tandis que les valeurs utilisées dans la spécifications sont les entiers non signés de 8 bits, définis dans la bibliothèque standard de F⋆ (FStar.UInt8.t), les valeurs utilisées dans l'API Low⋆ sont d'un type particulier (Hacl.UInt8.t), modélisant également des valeurs entières non signées sur 8 bits mais différents de FStar.UInt8.t. Ils sont en effet considérés comme secrets et se voient imposés un certain nombre de restrictions :

1. il n'est pas possible de les comparer (l'opérateur de comparaison '=' en F⋆ n'est pas implémenté pour eux, ni aucun opérateur booléen de comparaison) ;

2. les opérateurs arithmétiques considérés comme ne s'exécutant pas en temps constant ne sont pas implémentés ; il s'agit notamment des opérateur '/' et '%', inversement la soustraction est acceptée ;
3. ils ne peuvent pas servir d'index pour un accès à un tableau ou pour effectuer de l'arithmétique de pointeurs ;
4. pour remplacer les opérateurs de comparaison des opérateurs de masquage sont implémentés, qui retournent des octets du même type (0x00 pour 'Faux', et 0xff pour 'Vrai').

Le mécanisme de typage associé garantit par ailleurs que les valeurs publiques puissent être converties en valeurs privées dans le code Low⋆, mais non l'inverse. La fonction de conversion d'une valeur privée en une valeur publique est dite *Ghost*, c'est à dire qu'elle est forcément effacée par F⋆ avant la compilation et ne peut donc servir que dans les spécifications et les preuves. Elle est implicitement utilisée lors des appels mem[buf], figure 3, ligne 11, afin de pouvoir comparer le résultat concret à celui de la spécification.

Tous les opérateurs autorisés sur les valeurs secrètes étant considéré comme exécutés en temps constant, et les comparaisons entre valeurs secrètes étant exclues, cette méthodologie garantit l'indépendance entres les valeurs secrètes d'entrée, et les traces d'exécution du programme. En admettant qu'un attaquant ne soit pas en mesure de distinguer l'exécution des opérateurs autorisés sur les valeurs secrètes sur deux valeurs différentes, il n'est pas en mesure de distinguer si deux traces d'exécution du programme correspondent à des valeurs identiques ou différentes.

Le code C généré La bibliothèque fournit un snapshot du code C déjà généré de façon à permettre à un utilisateur de s'approprier le code directement, sans avoir à installer le système F⋆. Le code C offre la même API que le code Low⋆. À titre d'exemple le prototype généré pour l'API Low⋆ présentée précédemment est décrit dans la figure 4.

```
void Hacl_Curve25519_crypto_scalarmult(uint8_t *output, uint8_t *secret, uint8_t *point);
```

Fig. 4. Prototype C de la fonction d'exponentiation de Curve25519 dans HACL⋆

Grâce aux garanties obtenues via le mécanisme de preuve et de génération de code de F⋆ et KreMLin, les propriétés prouvées pour le code Low⋆ (sûreté mémoire, correction fonctionnelle et indépendance vis-à-vis des secrets) sont préservées dans le code généré. Par ailleurs un effort tout particulier a été fait au niveau du générateur de code afin que le code

produit soit de bonne qualité, c'est à dire qu'il puisse être audité par un industriel afin de le mettre en production dans sa propre bibliothèque.

2.4 Aucun compromis sur les performances

Parce que Low⋆, le fragment de F⋆ utilisé pour l'implémentation du code optimisé est très proche du fragment de C utilisé pour implémenter la cryptographie en général, les performances du code obtenu sont identiques sinon meilleures que celles des implémentations originales donc HACL⋆ s'est inspirée. En effet, pour certains algorithmes il apparaît que la version de référence était plus prudente que nécessaire et on prouve formellement qu'une retenue existant dans le code de référence par exemple, n'est pas nécessaire à la correction de la primitive.

Algorithm	**HACL***	**OpenSSL**	**libsodium**	**TweetNaCl**	**OpenSSL (asm)**
SHA-256	13.43	16.11	12.00	-	7.77
SHA-512	8.09	10.34	8.06	12.46	5.28
Salsa20	6.26	-	8.41	15.28	-
ChaCha20	6.37 (ref) 2.87 (vec)	7.84	6.96	-	1.24
Poly1305	2.19	2.16	2.48	32.65	0.67
Curve25519	154,580	358,764	162,184	2,108,716	-
Ed25519 sign	63.80	-	24.88	286.25	-
Ed25519 verify	57.42	-	32.27	536.27	-
AEAD	8.56 (ref) 5.05 (vec)	8.55	9.60	-	2.00
SecretBox	8.23	-	11.03	47.75	-
Box	21.24	-	21.04	148.79	-

Tableau 1. Intel64-GCC : Comparaison de performance en cycles/octet pour un Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz opérant sous Debian Linux 4.8.15 64-bit. Toutes les mesures (sauf Curve25519) sont basées sur des messages de 16KB ; pour Curve25519 le nombre indiqué correspondance au nombre de cycles CPU pour une seule exponentiation modulaire ECDH. L'ensemble du code a été compilé avec GCC 6.3. La version de OpenSSL est 1.1.1-dev (avec l'option `no-asm`) ; celle de Libsodium est 1.0.12-stable (avec l'option `--disable-asm`) et celle de TweetNaCl : 20140427.

À titre d'exemple, dans le code de Curve25519 Donna[6] écrit par Adam Langley, l'un des principaux développeurs de primitives cryptographique chez Google, la séquence d'opérations de retenues faisant intervenir `r1` et `r2`, lignes 137 et 138 de la figure 5 n'est pas nécessaire, et peut être omise.

[6] `https://github.com/agl/curve25519-donna/blob/master/curve25519-donna-c64.c`

```
129 t[3] += ((uint128_t) r4) * s4;
130
131             r0 = (limb)t[0] & 0x7ffffffffffff; c = (limb)(t[0] >> 51);
132 t[1] += c; r1 = (limb)t[1] & 0x7ffffffffffff; c = (limb)(t[1] >> 51);
133 t[2] += c; r2 = (limb)t[2] & 0x7ffffffffffff; c = (limb)(t[2] >> 51);
134 t[3] += c; r3 = (limb)t[3] & 0x7ffffffffffff; c = (limb)(t[3] >> 51);
135 t[4] += c; r4 = (limb)t[4] & 0x7ffffffffffff; c = (limb)(t[4] >> 51);
136 r0 += c * 19; c = r0 >> 51; r0 = r0 & 0x7ffffffffffff;
137 r1 += c; c = r1 >> 51; r1 = r1 & 0x7ffffffffffff;
138 r2 += c;
```

Fig. 5. Extrait de Curve25519-donna64

3 Le processus industriel et la mise en production dans Firefox

Comme la plupart des développeurs en charge du maintien des bibliothèques cryptographiques existantes, l'équipe de NSS a pris conscience de la difficulté d'introduire de nouvelles constructions cryptographiques dans sa bibliothèque et de s'assurer de la fiabilité du code existant à l'aide des méthodologies de développement « classiques ». En particulier, les tests unitaires sont clairement insuffisants car il est impossible de couvrir entièrement les grands espaces des états en entrée des primitives cryptographiques. Les méthodes formelles ayant fait d'importants progrès ces dernières années, celles-ci se sont, de facto, imposées comme étant la meilleure solution pour Mozilla afin d'accroître le niveau de confiance dans cette bibliothèque.

3.1 Les conditions nécessaires à une mise en production

Les méthodes formelles, particulièrement celles ayant récemment émergé du monde académique, doivent fournir certaines garanties aux industriels afin de pouvoir être utilisées en production. Parmi celles-ci nous pouvons identifier :

1. les performances du code ;
2. la lisibilité et la facilité d'audit du code par une tierce partie ;
3. la validation et le passage des tests existants ;
4. l'intégration de la méthodologie dans le *workflow* existant ;
5. la compatibilité avec l'ensemble des plateformes et architectures supportées.

Plusieurs mois ont été nécessaires afin de réaliser les changements nécessaires à l'intégration du code dans Firefox. Cependant les autres cas d'utilisation du code confirment la plupart de ces besoins.

3.2 Performance

Les primitives cryptographiques sont nécessaires à la sécurité des protocoles de communication tels que TLS et à leurs performances. Pour de nombreux usages industriels, les performances sont un point critique quant à l'adoption ou non d'un composant logiciel. En particulier, dans des domaines où les performances du code vont directement affecter l'expérience utilisateur, il est difficilement défendable auprès de ceux-ci d'obtenir un surcoût de sécurité au prix de performances dégradées. La performance est donc de nos jours un argument de vente essentiel pour de nombreux produits.

Dans le cadre de l'intégration de HACL* dans Firefox, nous avons choisi comme premier algorithme la version 64 bits de X25519 [1], la multiplication scalaire sur la courbe elliptique Curve25519 [4]. Plusieurs raisons ont conditionné ce choix, notamment le fait que la version C du code généré dans HACL* pour cet algorithme a d'excellentes performances. À notre connaissance la multiplication scalaire de notre code est la plus rapide des versions C non vectorisées (donc portables) existantes, et est 20% plus rapide que le code existant dans NSS. De manière peu étonnante, même si Mozilla n'utilise cet algorithme que du côté client dans Firefox, certains utilisateurs utilisent NSS côté serveur ce qui induit une économie en besoins matériels non négligeable.

Il est intéressant de noter que, dans le cas des algorithmes cryptographiques asymétriques, cette contrainte de performance est moins présente que pour les algorithmes symétriques qui sont utilisés intensivement au cours d'une grande partie des connexions au web, typiquement avec TLS.

3.3 Lisibilité et facilité de l'audit du code

Les méthodes formelles et les langages de programmation orientés vérification peuvent être de manière générale plus difficiles à appréhender que les langages de programmation traditionnels. Ceci provient notamment du fait que, dans certains langages comme Low*, les éléments de preuves peuvent parfois polluer la lecture du code fonctionnellement utile. D'autres langages comme Gallina [3] (Coq) séparent la preuve du code, ce qui présente à l'inverse l'inconvénient de ne pas aider à corréler les appels aux fonctions avec les appels aux tactiques qui fournissent les preuves.

Contrairement à la plupart des développements académiques, les composants logiciels industriels nécessitent une lisibilité élevée et une grande homogénéité stylistique dans des modèles bien définis. Le code C produit par la chaîne d'extraction de F* et KreMLin se doit donc d'être à la fois

beau et *lisible* ; l'objectif principal étant de se rapprocher autant que faire se peut du code qu'un expert aurait produit.

3.4 Multiplication des tests

Un ensemble de tests est effectué sur les spécifications de haut niveau en F⋆, le code optimisé en Low⋆ et le code C généré. En effet, les spécifications elles-mêmes sont extraites vers du code OCaml, compilées et passent des tests unitaires. Le code en Low⋆, à travers le code C généré, doit également passer des tests.

Ces tests comprennent les vecteurs de tests usuels contenus, par exemple, dans les spécifications textuelles des primitives (RFCs, spécifications FIPS du NIST) mais aussi d'autres vecteurs de tests aléatoires pour lesquels les valeurs de sorties sont comparées avec celles produites par d'autres bibliothèques de référence. Enfin, Mozilla possédant sa propre chaîne de tests, les primitives de HACL⋆ sont finalement éprouvées de nouveau via les tests fonctionnels de NSS puis de Firefox qui contiennent des essais de connections TLS, des tests d'APIs de haut-niveau, de nouveaux tests unitaires, d'interopérabilité...

3.5 Intégration des méthodes formelles dans le workflow existant

Plusieurs méthodologies s'affrontent dans n'importe quel projet introduisant des méthodes formelles. Il est possible de changer complètement le *workflow* de développement, cependant pour un projet aussi large, ce processus n'a pas été retenu. Lors de la phase de mise en production de HACL⋆ dans NSS, la question suivante s'est posée : « Comment gérér le fait que la chaîne de vérification est complexe et quelle confiance doit-on lui accorder ? ». L'importance du compromis dans des projets aussi novateurs est réellement primordial. Un positionnement potentiel pourrait possiblement être de faire confiance aveuglément à la chaîne d'extraction et d'effectuer un audit de la spécification des algorithmes uniquement puis d'intégrer directement le code C fourni par nos soins. Il est cependant raisonnable de concevoir que Mozilla puisse vouloir deux choses :

1. auditer directement le code C ;
2. avoir un contrôle indépendant des outils de vérification.

La solution trouvée consiste à procéder à un audit direct du code C mais aussi de la spécification ; en effet, c'est bien en premier lieu le C qui est mis en production, il se doit donc d'être évalué indépendamment de

la chaîne d'extraction. L'idée à moyen terme derrière cette stratégie est de donner de plus en plus de confiance à la chaîne d'extraction afin d'en arriver éventuellement à effectuer la revue de la spécification uniquement.

La chaîne de vérification et d'intégration actuelle La figure 6 présente le workflow de vérification intégré en amont des tests dans la chaîne d'intégration continue de NSS. La totalité des étapes permettant la vérification et la génération de code pour NSS y est incluse et fonctionne en permanence dans le CI de Mozilla.

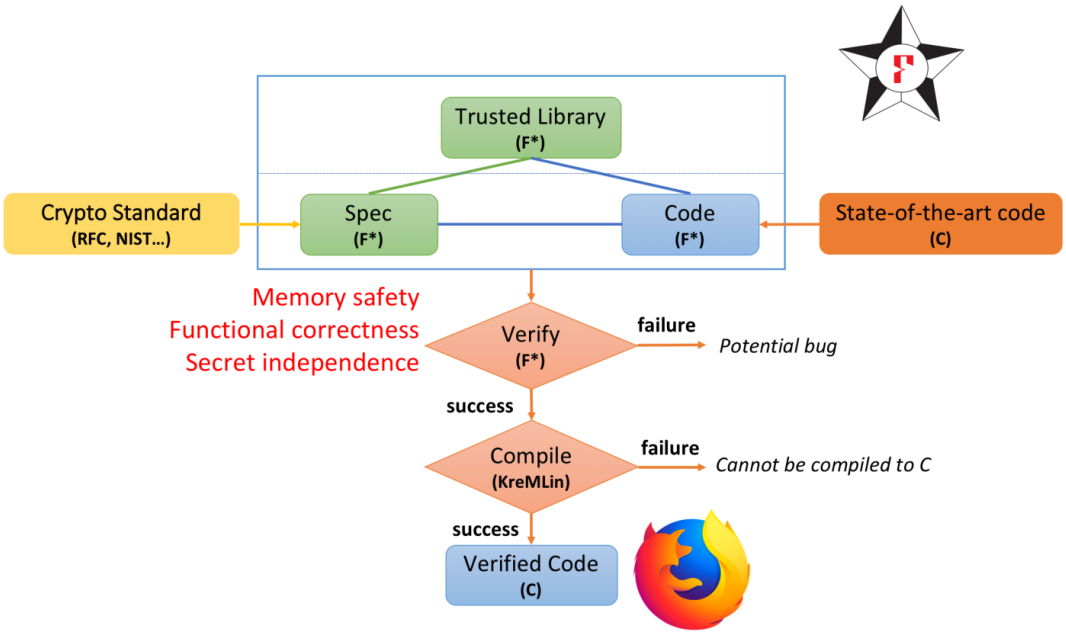

Fig. 6. Méthodologie de verification de HACL⋆

Cette méthodologie permet à Mozilla de vérifier en permanence que les outils continuent de fonctionner et leur permet d'avoir la maîtrise de la chaine dans le cas où ils souhaiteraient modifier eux-même le code afin de l'optimiser ou d'y ajouter de nouvelles primitives. De plus cette intégration permet continuellement d'incrémenter le *commit* de référence pour HACL⋆ et ainsi de bénéficier de l'ensemble des progrès apportés à nos outils, comme par exemple les améliorations de l'ensemble de l'extraction pour la beauté et l'efficacité du code C. Une utilité importante de cette intégration est également de vérifier à chaque *commit* dans NSS que le code vérifé n'a pas été altéré de manière malveillante ou accidentelle. En

effet, si le code issu de l'extraction est différent du *commit* en question, l'intégration continue signalera que le code poussé n'est pas vérifié.

4 Conclusions

HACL⋆ démontre que l'application de la méthodologie choisie avec F⋆ permet une utilisation à l'échelle de centaines de millions d'utilisateurs des méthodes formelles. Désormais de nombreux algorithmes tels que X25519, Chacha20 ou Poly1305 sont en production dans Firefox, après avoir suivi la méthodologie présentée dans cet article. Il est évident que la confiance et les gros efforts de dialogue et le travail sur l'accessibilité des outils pour l'utilisation par des industriels, certes avertis, payent. La cryptographie reste un champ d'application restreint mais nous pensons que notre méthodologie actuelle peut passer à l'échelle. Les projets mettant en relation les industriels et les chercheurs au niveau d'internet et du web sont relativement peu nombreux, particulièrement quand ils sont liés aux méthodes formelles. Il est important d'en prendre soin, en renforçant les liens entre Mozilla, Inria et Microsoft, et en les valorisant sur le long terme. Le web ne peut que mieux s'en porter ! QED.

Références

1. Elliptic Curves for Security. IETF RFC 7748, 2016.
2. Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra Monads for Free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 515–529. ACM, January 2017.
3. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. Projet COQ.
4. Daniel J Bernstein. Curve25519 : new Diffie-Hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.
5. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Catalin Hritcu, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Verified Low-Level Programming Embedded in F*. arXiv :1703.00053, to appear at ICFP 2017, 2017.
6. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3 : An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
7. Heartbleed. The Heartbleed Bug. `http://heartbleed.com/`, 2014.

8. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.
9. OpenSSL library. OpenSSL : Cryptography and SSL/TLS Toolkit, 1998–2017.
10. Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, 2016.
11. Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.