Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec)

Titre	**Recommandations relatives à l'utilisation du langage OCaml et à l'installation et la configuration des outils associés**
Identifiant	**Livrable L3.3**
Version	**7.0**
Date	**2012-12-07**
Pages	**68**
Approbation	**Christèle Faure, SafeRiver**
	Date: Signature:

Table des révisions

Version	Date	Description et changements	Parties modifiées
1.0	2011-04-15	Version initiale correspondant au livrable L3.3.1	Tout le document
2.0	2011-05-10	Version incorporant les commentaires de l'ANSSI	Tout le document
3.0	2011-10-14	Version correspondant à la première fourniture du livrable L3.3.2	Tout le document
4.0	2011-10-28	Version incluant les remarques de l'ANSSI	Tout le document
5.0	2011-11-28	Version prenant en compte les remarques de l'ANSSI	Tout le document
6.0	2012-11-05	Version correspondant au livrable L7.3	Ajout du chapitre 7 de mise à jour
7.0	2012-12-07	Version intégrant les mises à jour	Tout le document, retrait du chapitre 7

Résumé

Ce document, réalisé dans le cadre du projet LaFoSec, propose un ensemble de recommandations relatives à la sécurité des applications développées en OCaml. Ce langage appartient à la famille des langages fonctionnels et offre également des traits impératifs, des traits objet et des modules. Ce document propose des recommandations portant sur l'utilisation du langage OCaml, son installation et la configuration des outils associés, recommandations visant à améliorer la robustesse des applications développées.

Table des matières

Introduction

Objet du document

Ce document a été produit dans le cadre de l'étude LaFoSec, relative au marché n° 2010027960021207501 notifié le 8 novembre 2010 par le Secrétariat Général de la Défense et de la Sécurité Nationale (SGDSN). Il liste les recommandations relatives à l'utilisation du langage OCaml et à l'installation et la configuration des outils associés (identifiant L3.3 dans le CCTP).

Présentation du projet LaFoSec

Les langages de programmation dits *fonctionnels* sont réputés offrir de nombreuses garanties facilitant le développement de logiciels soumis à des exigences de sûreté ou de sécurité. Par exemple, la société *Ericsson* a développé le langage fonctionnel Erlang, dédié à la concurrence, pour ses applications de communication. La société *Esterel Technologies* a développé le langage fonctionnel SCADE, dédié au traitement synchrone de flots de données, pour le traitement de logiciel critique.

Dans le cadre de ses activités d'expertise, l'ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information) souhaite bénéficier d'une assistance scientifique et technique pour déterminer l'adéquation de ces langages au développement d'applications de sécurité et disposer d'une étude permettant d'améliorer la confiance vis-à-vis de la sécurité de ces applications. C'est l'objet du projet LaFoSec.

Le projet LaFoSec consiste en une étude prospective des langages fonctionnels, visant à déterminer les caractéristiques propres à ces langages susceptibles de répondre aux exigences de sécurité.

Tout d'abord, les caractéristiques des langages fonctionnels sont décrites et plusieurs langages généralistes sont présentés selon ces caractéristiques. Ensuite, cette étude est approfondie en se restreignant à trois langages, OCaml, F$^{\#}$ et

Scala, pour recenser leurs avantages et inconvénients du point de vue de la sécurité.

Présentation du contexte

Ce document recense les recommandations permettant d'augmenter la sécurité des applications développées avec le langage OCaml. Le document renvoie aux livrables [ANA-SECU, 2011] et [MODE-EX, 2011] de l'étude LaFoSec pour les détails de l'analyse de sécurité du langage OCaml et de son modèle d'exécution. Une mise à jour de ce document a été effectuée suite au développement de l'application de la Tranche Conditionnelle n° 1 (TC1) et de l'évaluation de sécurité de cette application réalisée lors de la Tranche Conditionnelle n° 2 (TC2). Les résultats de l'évaluation sont présentés dans le livrable [TC2-RAP, 2012].

Organisation des recommandations

Les réponses apportées par le langage OCaml aux nécessités de sécurité des applications ont été étudiées dans les documents [ANA-SECU, 2011] et [MODE-EX, 2011] qui contiennent des avis de sécurité classés en intérêts, dangers et recommandations. Ces éléments sont repris dans ce document sous forme de recommandations de sécurité destinées aux développeurs d'applications OCaml. Les recommandations usuelles sur la qualité du développement logiciel ne sont en revanche pas traitées.

La structure de ce document s'appuie sur un cycle classique en V de développement logiciel : les dangers et recommandations émis dans les documents [ANA-SECU, 2011] et [MODE-EX, 2011] sont regroupés par étape de développement pour laquelle ils sont pertinents et par thème. Une fiche thématique liste des recommandations précédées d'une courte explication ainsi que les références aux documents dont elles sont issues.

La numérotation des recommandations suit la forme suivante : R-i-j où i est le numéro de la fiche à laquelle elle appartient et j le numéro unique de la recommandation. Une même recommandation peut figurer dans plusieurs fiches, elle porte alors plusieurs numéros mais une référence croisée signale le lien.

Plan du document

Les chapitres 1, 2 et 3 portent sur les étapes d'architecture, de conception détaillée et de production du texte source.

L'étape de production de code objet est traitée dans le chapitre 4, il présente donc les recommandations à suivre pendant la compilation.

Le chapitre 5 contient les recommandations relatives à l'exécution. Les recommandations pour l'installation du système OCaml et pour la configuration des outils associés au langage sont indiquées dans le chapitre 6.

Chapitre 1

Architecture

La phase de définition de l'architecture d'un système logiciel a pour but de définir le système comme un assemblage de sous-systèmes, dont les interactions sont déterminées, les sous-systèmes eux-mêmes étant (le plus possible) indépendants.

Fiche 1 **Architecture — modules et interfaces**

Le mécanisme de modules de OCaml permet de refléter la décomposition d'un système logiciel en sous-systèmes développés indépendamment, les communications entre sous-systèmes étant reflétées par les interfaces de ces modules. Le découpage en modules compilables séparément garantit l'indépendance entre les sous-systèmes eux-mêmes et la conformité des interfaces aux interactions requises.

R-1-1 Utiliser le système de modules pour représenter l'architecture du logiciel, en introduisant un module par sous-système.

Il est possible de contrôler la pertinence de la définition d'un module vis-à-vis de sa spécification. En effet, certains éléments de la spécification d'un module peuvent être exprimés dans la définition (a priori) de l'interface du module à développer. Or, si aucune interface n'est fournie, la compilation du module infère une interface par défaut, qui décrit toutes les fonctionnalités offertes par le module. Cette interface par défaut peut être confrontée à l'interface souhaitée pour vérification de conformité.

R-1-2 Écrire d'abord l'interface d'un module avant de définir le module lui-même. Pour plus de garanties, comparer cette interface avec celle inférée par la compilation du corps du module (option `-i`).

Fichiers d'interfaces et interfaces multiples

Un fichier `.ml` représente le corps d'un module. Si un fichier `.mli` de même nom existe, il représente l'interface de ce module. Sinon, le compilateur utilise l'interface inférée qui exporte tous les éléments du module.

R-1-3 Associer à chaque fichier `.ml` un fichier `.mli`.

Le mécanisme décrit ci-dessus ne permet pas d'associer plusieurs interfaces différentes à un même module.

Il est possible de définir plusieurs interfaces pour un même module en définissant des sous-modules d'un fichier `.ml`. Ainsi dans l'exemple ci-dessous, un fichier d'implémentation `f.ml` définit un module `M` et deux versions de ce module utilisant deux interfaces différentes.

```
type t = string
module M = struct ... end
module Complet : sig
  val lire : t -> string
  val creer : string -> t
end = M
module Restreint : sig
  val lire : t -> string
end = M
```

Un fichier d'interface `f.mli` n'exporte que les deux modules ainsi définis.

```
type t
module Complet : sig
  val lire : t -> string
  val creer : string -> t
end
module Restreint : sig
  val lire : t -> string
end
```

Les sous-modules du module `F` peuvent être utilisés selon les besoins requis. Les données restent compatibles car le type `t` est partagé entre les deux modules.

```
1 F.Restreint.lire (F.Complet.creer "abc");;
2 - : string = "abc"
```

Il est souhaitable, tout au long du cycle de développement, de définir les interfaces des modules de manière à répondre seulement aux besoins des modules importateurs. Il en est de même en cas de réutilisation d'un module déjà défini.

R-1-4 Définir, pour un module exportateur, une interface adaptée aux besoins de chaque module importateur.

Noms des modules

R-1-5 Choisir des noms de modules significatifs et distincts. (analogue à **R-12-65**)

R-1-6 Ne jamais masquer les noms des modules de la bibliothèque standard. (analogue à **R-12-64**)

Références externes

– [ETAT-LANG, 2011, §1.1.5] (Modules)
– [ANA-SECU, 2011, §1.3] (Traits modulaires)

Fiche 2 **Architecture — classes**

Les classes, comme les modules, pourraient a priori être conseillées pour représenter l'architecture d'un système. Cependant, les besoins en encapsulation des applications ayant trait à la sécurité sont importants et seuls les modules fournissent un mécanisme d'encapsulation sûr. Or les classes ne peuvent pas contenir de modules alors qu'une classe est nécessairement définie dans un module avec ou sans interface. Afin de préserver les possibilités d'encapsulation, il est donc préférable de n'utiliser que des modules pour représenter l'architecture d'un système.

R-2-7 Ne pas utiliser les classes pour traduire la décomposition en sous-systèmes.

La fiche 5 fournit d'autres recommandations concernant la conception objet.

Références externes

– [ANA-SECU, 2011, §1.4] (Traits objet)

Fiche 3 **Architecture — choix d'implantation**

Choix du modèle d'implantation

Les recommandations sur les modèles d'implantation sont expliquées et détaillées dans les chapitres 4 et 5. Ne sont reprises ici que les recommandations devant être prises en compte dès la conception du système.

Un exécutable en code natif peut offrir des propriétés de sécurité plus fiables que celles fournies par un exécutable en bytecode, produit à partir du même texte source. Le modèle natif positionne le code dans la zone en lecture seule (si le système d'exploitation le permet).

Si la spécification de l'application nécessite la compilation vers du bytecode ou si l'application doit s'exécuter sur une architecture non ciblée par le compilateur natif, la compilation en bytecode est la seule solution.

R-3-8 Utiliser de préférence le modèle d'implantation en code natif. Justifier toute implantation en bytecode. Utiliser de préférence le mode `-custom` et sinon, justifier le choix.

R-3-9 Ne jamais utiliser l'exécution par boucle interactive en exploitation.

R-3-10 En exploitation, ne jamais utiliser les options de débogue et de profilage du compilateur.

(analogue à **R-21-122**, **R-23-128** et **R-25-137**)

En bytecode, certains comportements associés au mode débogue d'exécution ne peuvent pas être désactivés, voir la fiche 23 pour plus de détails.

Compilation séparée

R-3-11 Placer les modules correspondant à chacun des sous-systèmes dans des fichiers compilables séparément.

Références externes

- [MODE-EX, 2011, §1.5.1] (Allocation, désallocation)
- [MODE-EX, 2011, §2.3.2] (*Backtrace*, pile de levée d'exception)
- [MODE-EX, 2011, §3.3.3] (Mode débogue)
- [MODE-EX, 2011, §5.3] (Différences d'exécution)

Fiche 4 **Architecture — bibliothèques**

Utilisation de bibliothèques préexistantes

Des bibliothèques préexistantes peuvent être disponibles sous la forme de textes source ou sous la forme de codes pré-compilés. Il est possible de les réutiliser sous certaines conditions.

La bibliothèque standard contient quelques modules dits non sûrs, dont l'utilisation est interdite (cf. fiche 16). Tous les autres modules peuvent être utilisés en s'assurant qu'ils proviennent bien de la distribution officielle et n'ont pas été altérés.

R-4-12 Proscrire l'utilisation des modules `Obj`, `Marshal` et `Printexc` dans le texte source de l'application.

Il existe également de nombreuses bibliothèques OCaml distribuées sur différents sites. Avant de les utiliser, il est nécessaire d'effectuer une analyse, une relecture de code, afin de s'assurer qu'elles ne font pas appel à des constructions non sûres et qu'elles ne contiennent pas de code malveillant. Il est également impératif de les recompiler. Toutes ces recommandations sont recensées dans la fiche 20 et résumées dans la recommandation suivante.

R-4-13 Vérifier le texte source et le processus de compilation des bibliothèques OCaml préexistantes et les recompiler.

Utilisation de bibliothèques externes

OCaml permet l'utilisation de codes externes natifs, chargeables par la construction `external` et provenant de sources C, ou de tout autre langage (voir la fiche 18). Ces codes externes ne peuvent pas être vérifiés automatiquement et doivent donc être, par défaut, considérés comme non sûrs.

R-4-14 Ne pas utiliser de codes externes, sauf pour des raisons dûment justifiées et documentées. (analogue à **R-18-94**)

R-4-15 Tout code externe utilisé doit être vérifié et, si possible, recompilé. La démonstration de son innocuité doit être faite.

Il est bien connu que le chargement dynamique de code peut introduire des vulnérabilités comme l'injection de code malveillant.

R-4-16 Si l'utilisation de code externe ne peut être évitée, construire l'exécutable uniquement par chargement statique de ce code.

Il peut cependant s'avérer nécessaire de charger dynamiquement du code externe. Dans ce cas, mettre en œuvre en plus de cette fiche les recommandations de la fiche 19.

Références externes

– [MODE-EX, 2011, §1.2.5] (Constructions invalidant l'analyse statique)

Chapitre 2

Conception détaillée

Modules et classes d'OCaml peuvent constituer de bons choix pour la phase de conception détaillée. Le système de modules offre une excellente lisibilité et un contrôle important du texte source. Le mécanisme de classes permet d'adopter un style de développement progressif, permettant d'implémenter étape par étape des spécifications de grande ampleur.

Les interfaces des modules et des classes introduisent les noms des éléments exportés et leur type. Les recommandations sur les types eux-mêmes sont présentées dans la fiche 6, qui rassemble les points utiles à la conception détaillée, d'autres recommandations sur les types étant fournies dans le chapitre consacré à la production du texte source (fiche 13 du chapitre 3).

Fiche 5 **Conception objet**

Le mécanisme de classes favorise une forme de partage de code grâce à la généricité offerte par l'héritage. La redéfinition apporte de la souplesse en offrant la possibilité d'adapter un traitement générique dans une partie spécifique du code. Cependant, le mécanisme de classes est plus complexe à gérer que celui des modules, la redéfinition et la liaison retardée permettant de faire évoluer la sémantique de méthodes définies antérieurement (même si cette évolution est recherchée).

R-5-17 Préférer les modules aux classes, lorsque l'ampleur du développement ou les besoins de réutilisation ne nécessitent pas l'utilisation de traits objet.

État interne d'un objet et visibilité des méthodes

Un objet est constitué d'une part de variables d'instance auxquelles il est le seul à pouvoir accéder, elles servent à définir (si besoin) son état interne. Il possède d'autre part des méthodes (marquées `private`) qu'il est le seul à pouvoir invoquer et enfin des méthodes pouvant être invoquées par tout autre objet. Ces différences d'accès permettent de bien clarifier le rôle et l'utilisation des champs d'un objet. Mais elles ne sont pas conçues pour répondre à des exigences de confidentialité et d'intégrité car elles ne sont pas systématiquement propagées par l'héritage et la redéfinition (voir la fiche 7).

Le marqueur `private`*, positionné dans le corps d'une classe, ne signifie pas que la méthode ainsi marquée soit invisible. La marque* `private` *disparaît en cas de redéfinition (par héritage).*

R-5-18 Ne pas utiliser les variables d'instance ou le marqueur `private` pour répondre à des exigences de sécurité.

R-5-19 Toute classe contenant des données sensibles doit être encapsulée dans un module associé à une interface. Celui-ci doit interdire tout héritage en n'exportant pas la classe et en ne donnant accès qu'aux méthodes reconnues sûres. (analogue à **R-8-36**)

Héritage et redéfinition

Les mécanismes d'héritage et de redéfinition permettent la construction d'une application étape par étape. Toutefois, si l'arborescence d'héritage est trop importante ou les redéfinitions trop nombreuses, le texte source peut devenir difficile à relire.

R-5-20 Limiter l'héritage à une profondeur raisonnable.

Les redéfinitions peuvent être explicitées dans le texte source en utilisant les mots-clés `val!` et `method!` pour redéfinir une variable d'instance ou une méthode. Des options de compilation permettent de contrôler ces redéfinitions (avertissement 7, voir la fiche 21).

R-5-21 Utiliser toujours la syntaxe explicite des redéfinitions ainsi que le contrôle de celles-ci.

Typage des classes et des objets

En OCaml, types de classe et types d'objet sont deux notions différentes.

*La déclaration d'une classe introduit le nom de la classe. Celui-ci désigne aussi le type de la classe (*class type*), il est en même temps un alias pour le type fermé des objets de cette classe et il dénote également le constructeur de cette classe.*

Fonctions sur les objets

Une fonction peut prendre un objet en paramètre. L'inférence de type attribue un type ouvert à ce paramètre. Si le type de cet objet est explicité dans la définition de la fonction (par mention du nom de sa classe), alors son type est fermé. Dans les deux cas, à cause du sous-typage, tout objet, quelle que soit sa classe, possédant les méthodes nommées dans le type objet demandé (avec concordance des types de ces méthodes), sera accepté comme paramètre effectif.

Même si le type de l'objet paramètre est fourni par le nom de la classe dans la définition de la fonction, cela ne signifie pas que cette fonction ne s'applique qu'aux objets de cette classe.

R-5-22 Utiliser les noms de méthode ou leurs types pour différencier des familles d'objets.

R-5-23 Encapsuler la classe et toute fonction qui ne s'applique qu'aux objets de cette classe dans un module sans exporter la classe.

Initialiseurs

À chaque création d'un objet par la construction `new`, l'initialiseur de la classe (indiqué par `initializer`) est exécuté. Il est à noter que l'initialiseur est transmis par héritage.

R-5-24 En cas d'utilisation de classes provenant de bibliothèques externes, contrôler non seulement les méthodes mais aussi les initialiseurs de classes.

Références externes

- [ETAT-LANG, 2011, §1.2.3] (Traits objet)
- [ANA-SECU, 2011, §1.4] (Traits objet)
- [Leroy *et al.*, 2010, Ch. 3] (*Objects in Caml*)

Fiche 6 **Langage de types**

OCaml offre un langage de types très expressif, permettant de définir très commodément n'importe quelle représentation de données sans manipulation de pointeurs (voir la fiche 13). Le système d'inférence de types permet au programmeur de ne pas avoir à déclarer les types des expressions qu'il manipule. Le typage apporte de nombreuses garanties sur la cohérence globale du programme.

L'égalité entre types est à la base de l'algorithme de typage : aucune valeur ne peut appartenir à deux types différents. Mais un type est égal à tous ses alias donc toute valeur d'un type est aussi une valeur de tous ses alias. Cela peut conduire à des erreurs de programmation difficilement détectables, si le programmeur a cru pouvoir utiliser des alias pour distinguer des sous-ensembles de valeurs d'un même ensemble.

R-6-25 Justifier les utilisations d'alias de type.

Dans l'exemple suivant, le type `a1` est un alias du type `int`, ses valeurs sont confondues avec les entiers. Le type `a2` introduit un constructeur `A` qui différencie ses valeurs.

```
1 type a1 = int
2 type a2 = A of int
3 let f1 (x:a1) = x + 1;;
4 val f1 : a1 -> int = <fun>
5 let f2 (x:a2) = x + 1;;
6 Error: This expression has type a2 but an expression ↵
    was expected of type int
```

Si deux familles de données appartenant à un même ensemble doivent recevoir des traitements différents, il est conseillé d'introduire un constructeur de valeur pour chacune d'elles afin de bénéficier des vérifications faites par le typage et le filtrage sur la séparation des traitements.

R-6-26 Séparer les sous-ensembles d'un même ensemble en introduisant, par une définition de type somme, un constructeur de valeur pour chacun de ces sous-ensembles.

Un type peut être défini par l'utilisateur sous forme d'un type somme ou enregistrement. Un tel type n'est égal qu'à lui-même et à ses alias qui pourraient être introduits après sa définition.

R-6-27 Représenter les données de préférence avec des types somme et enregistrement.

L'exhaustivité du filtrage est vérifiable pour les types produit, enregistrement, somme (donc sur les types `list` et `option`). La fiche 14 détaille les recommandations concernant le filtrage.

R-6-28 Préférer les types permettant la vérification d'exhaustivité du filtrage.

Le langage de types permet de représenter l'absence de valeur significative par le type `'a option` (absence représentée par la valeur `None` et présence d'une valeur `v` par `Some v`). Cela évite d'utiliser une valeur par défaut, dont la pertinence peut être remise en cause par une évolution ultérieure du logiciel. La vérification d'exhaustivité du filtrage assure que les cas liés à une absence de valeur significative sont tous examinés.

R-6-29 Représenter l'absence de valeur significative à l'aide d'un type option.

Références externes

- [ETAT-LANG, 2011, §1.1.2] (Types et construction de données)
- [ANA-SECU, 2011, §1.1.3] (Étude du typage)
- [ANA-SECU, 2011, §1.1.6] (Récapitulatif)
- [MODE-EX, 2011, §1.2.2] (Analyse statique du texte source)
- [Leroy *et al.*, 2010, §6.8] (*Type and exception definitions*)

Fiche 7 Conception détaillée et sécurité

La confidentialité et l'intégrité des données sensibles et plus généralement les garanties de sécurité peuvent être renforcées en combinant l'utilisation de plusieurs traits de langage. Ces traits sont présentés dans différentes fiches. Cette fiche recense les principaux points.

La recommandation suivante est détaillée dans la fiche 8.

R-7-30 Le choix des interfaces des sous-systèmes doit assurer l'encapsulation des données sensibles.

Les fiches 16, 17, 18 et 19 présentent les constructions dangereuses et les différentes recommandations qui leur sont applicables. La recommandation suivante les résume.

R-7-31 Les constructions non sûres sont à prohiber dans le texte source.

La recommandation suivante résume celles de la fiche 13, consacrée à la représentation des données et celles de la fiche 11 en les particularisant aux données sensibles. Le cas des données qui doivent être effacées après usage sera traité dans la fiche 9.

R-7-32 Les données sensibles doivent être représentées par un type permettant égalité structurelle et filtrage. Tout autre choix doit être argumenté. Toute sous-structure mutable figurant dans le type choisi doit être justifiée.

Pour garantir leur confidentialité et leur intégrité, les valeurs de ces données doivent être enveloppées dans un type fonctionnel et encapsulées dans un module contenant les traitements afférents, dont l'interface sera soigneusement étudiée. Les recommandations des fiches 8 et 9 détaillent les recommandations sur l'enveloppement et l'encapsulation de ces données sensibles. Dans l'exemple suivant, `interne` est un type somme représentant des données sensibles, il est enveloppé et encapsulé dans un type abstrait `prive`.

```
1 module M : sig
2   type prive
3   val test : int -> prive -> bool
4   val val1 : prive
5   val val2 : prive
```

```
6  end = struct
7    type interne = A of int | B of bool * int
8    type prive = unit -> interne
9    let val1 = fun () -> A (25)
10   let val2 = fun () -> B (true, 36)
11   let test y x =
12     match x () with
13     | A (z) -> ...
14     | B (true, z) -> ...
15     | B (false, z) -> ...
16 end;;
```

Le compilateur de OCaml fournit un certain nombre de garanties sur les propriétés du code. L'utilisation de l'outil **camlp4** peut en apporter d'autres. La relecture des textes source utilisés dans l'application en cours de développement constitue l'ultime protection possible contre du texte source malveillant. Elle permet de contrôler que chacune des recommandations prescrites est bien respectée (voir la fiche 20).

R-7-33 Procéder à la relecture des textes source dont on ne peut affirmer l'innocuité. (analogue à **R-20-107**)

Références externes

– [ANA-SECU, 2011, §1.1.3] (Étude du typage)
– [MODE-EX, 2011, §1.2.5] (Constructions invalidant l'analyse statique)

Fiche 8 **Encapsulation**

L'encapsulation est un mécanisme qui restreint les possibilités de lecture, écriture ou exécution suivant les modalités de sa mise en œuvre. Si une unique valeur sensible doit n'être connue que d'une fonction (ou d'un petit groupe de fonctions), il est possible de l'encapsuler dans la fermeture de cette fonction (voir la fiche 10). L'application partielle d'une fonction à une fonction anonyme ou à une donnée permet également d'encapsuler du code ou une donnée.

Le module `Marshal` *et son option* `Closures` *permettent de briser l'encapsulation des fermetures (voir la fiche 16).*

Dans l'exemple suivant, la fonction `chiffreur_gen` demande une clé `k` à l'utilisateur et retourne une fonction de chiffrement correspondante. La clé `k` est encapsulée dans le chiffreur et n'est donc visible que par le corps de la fonction `chiffreur`.

```
1 let chiffreur_gen () =
2   Printf.printf "Entrez votre cle: ";
3   let k = read_int () in
4   fun m -> (m + k) mod 256;;
5 val chiffreur_gen : unit -> int -> int = <fun>
6 let chiffreur = chiffreur_gen ();;
7 Entrez votre cle: 54
8 val chiffreur : int -> int = <fun>
```

Mis à part ces moyens restreints mais efficaces, l'encapsulation se fait par la définition d'interfaces de modules (et non par des classes, voir ci-dessous). Si le nom d'une fonction, d'une variable ou d'un type ne figure pas dans l'interface, alors cet élément du module est inaccessible à l'extérieur du module. Si seul le nom d'un type figure dans l'interface, alors ce type (dit abstrait) peut être utilisé à l'extérieur mais sa structure reste inconnue et il n'est donc pas possible de construire ou analyser directement des valeurs de ce type. Si un type ne figure pas dans l'interface, aucune valeur de ce type ne peut être manipulée à l'extérieur du module, ce confinement est garanti par le typage.

R-8-34 Confiner dans un module les types des données sensibles ne devant pas être manipulées à l'extérieur du module ainsi que les traitements les concernant.

R-8-35 Abstraire les types des données sensibles définis dans un corps de module si ces données doivent être manipulées à l'extérieur du

module. Ne fournir dans l'interface du module que le nom du type et les types des fonctions nécessaires à ces manipulations.

Les classes fournissent également un mécanisme d'encapsulation avec leurs variables d'instance, qui ne sont accessibles que par l'objet lui-même. Ces variables pouvant être de n'importe quel type, elles peuvent servir à encapsuler à la fois du code et des données. Mais cette protection s'avère aisément contournable par une utilisation malveillante de l'héritage. Il ne faut donc pas se reposer sur l'encapsulation des variables d'instance d'un objet pour protéger des données ou du code sensibles.

R-8-36 Toute classe contenant des données sensibles doit être encapsulée dans un module associé à une interface qui interdit tout héritage et ne donne accès qu'aux méthodes reconnues sûres.

(analogue à **R-5-19**)

Contournement de l'encapsulation

Le contournement de l'encapsulation est rendu possible par les mécanismes suivants :

- L'observation de données encapsulées par utilisation conjointe des exceptions et de l'égalité (`=`, `!=`) ou de la comparaison (`<`, `>`, `<>`, `<=`, `>=`, `compare`), ou bien par utilisation des fonctions `hash` et `hash_param` de la bibliothèque de hachage `Hashtbl`. Une valeur sensible peut être identifiée en procédant par dichotomie ou par essais/erreurs (voir [ANA-SECU, 2011, §1.1.4]).
- Le contournement du typage appliqué à des données encapsulées, avec l'utilisation de constructions non sûres (voir la section 3.2).

Les valeurs de type fonctionnel (les fonctions) ne pouvant être comparées, il est possible d'envelopper des données sensibles dans un type fonctionnel pour renforcer leur encapsulation (voir [MODE-EX, 2011, §1.2.4]). Toutefois, la fonction de comparaison `compare` et l'opérateur `==` permettent d'identifier deux instances de la même fermeture et la fonction de hachage `Hashtbl.hash` permet de comparer des fermetures différentes.

R-8-37 Compléter la protection des données sensibles encapsulées dans des modules en les enveloppant dans un type fonctionnel.

L'inclusion dans un type fonctionnel permet de répondre au problème de l'observation de données encapsulées mais elle ne répond pas à celui de l'égalité physique sur les données encapsulées.

Dans l'exemple suivant des valeurs entières sont incluses dans un type fonctionnel encapsulé. Ainsi, il n'est pas possible de les observer par égalité ou comparaison (`M.f = M.h` lève une exception) mais il est toutefois possible d'identifier que deux valeurs sont construites avec une même fermeture (`compare M.f M.g`).

```
module M : sig
  type t
  val f : t val g : t val h : t
end = struct
  type t = { c : unit -> int }
  let f = { c = (fun () -> 3) }
  let g = { c = f.c }
  let h = { c = (fun () -> 3) }
end;;
module M : sig type t val f : t val g : t val h : t ↵
    end
M.f = M.g;;
Exception: Invalid_argument "equal: functional value".
M.f = M.h;;
Exception: Invalid_argument "equal: functional value".
compare M.f M.g;;
- : int = 0
compare M.f M.h;;
Exception: Invalid_argument "equal: functional value".
```

R-8-38 Contrôler les utilisations des fonctions et opérations d'égalité, de comparaison et de hachage sur les données comportant des sous-structures sensibles. (analogue à **R-9-42**)

Références externes

- [ANA-SECU, 2011, §1.1.4] (Mécanismes de gestion des expressions)
- [ANA-SECU, 2011, §1.3.2] (Encapsulation et interfaces de module)
- [MODE-EX, 2011, §1.2.4] (Comparaison et hachage)
- [MODE-EX, 2011, §1.2.5] (Constructions invalidant l'analyse statique)
- [Leroy *et al.*, 2010, §2.2] (*Signatures*)
- [Leroy *et al.*, 2010, §6.10] (*Module types (module specifications)*)

Fiche 9 Fuite de données confidentielles

Des données confidentielles peuvent fuir indirectement par observation illicite (cf. fiche 8) ou directement par transmission de leurs valeurs. La vérification de la portée des identificateurs des textes source d'un programme permet de vérifier l'absence de fuite directe de données.

La valeur d'un identificateur `x` ne peut fuir que de l'une des manières suivantes, facilement détectables par lecture du code :

- Appel d'une fonction faisant fuir son paramètre avec `x` en argument ;
- Affichage de la valeur de `x` ;
- Affectation de la valeur de `x` dans un mutable (cf. fiche 11) ;
- Retour de `x` à la fin de la portée de `x` ;
- Levée d'une exception avec `x` comme argument ;
- Liaison d'un nouvel identificateur `y` à la valeur de `x` et fuite de `y` par l'une des méthodes de cette liste.

R-9-39 Vérifier l'absence de fuite des données confidentielles par relecture du code.

R-9-40 Interdire le passage de données sensibles en paramètre d'exception, même si elles sont encapsulées. (analogue à **R-15-83**)

R-9-41 Vérifier que les bibliothèques utilisées ne font aucun passage de données sensibles en paramètre d'exception. (analogue à **R-15-84**)

R-9-42 Contrôler les utilisations des fonctions et opérations d'égalité, de comparaison et de hachage. (analogue à **R-8-38**)

La fiche 11 détaille les recommandations sur les données mutables et la fiche 26 porte sur leur mise en œuvre à l'aide du préprocesseur `camlp4`.

R-9-43 Justifier et contrôler soigneusement les utilisations de données mutables.

R-9-44 Proscrire les effets de bord dans les parties de code devant répondre à des exigences de sécurité. S'ils sont nécessaires, les encapsuler dans un module. (analogue à **R-11-54**)

Données confidentielles et GC

La gestion de la mémoire du programme est prise en charge par le Garbage Collector (GC) en OCaml.

Le GC peut dupliquer en mémoire vive des données confidentielles structurées et donc augmenter leur temps de présence dans la mémoire. Le GC n'effectue aucun effacement des données manipulées par le programme que ce soit lors de leur libération ou en fin d'exécution du programme.

Pour contrôler l'effacement de données confidentielles les plus sensibles (comme par exemple les clefs cryptographiques), il convient de les représenter par des données mutables simples allouées hors de l'espace mémoire géré par le GC (valeurs de types `Bigarray.char`, `Bigarray.int`, etc.), d'encapsuler dans un module la définition de cette structure mutable ainsi que toutes les opérations sur cette structure, et de n'exporter aucune fonction de modification, sauf une fonction d'effacement. Voir la fiche 11 concernant l'utilisation de données mutables et la fiche 8 concernant l'encapsulation.

R-9-145 Pour chaque type de donnée confidentielle sensible, encapsuler dans un module les fonctions qui manipulent ce type, fournir dans ce module une fonction d'effacement, et n'exporter le type de données que de manière abstraite.

R-9-146 Ne pas se reposer sur les fonctions de finalisation du GC pour effacer les données confidentielles.

La duplication des données confidentielles par le GC s'effectue en mémoire vive. Il est à noter que les mécanismes système peuvent dupliquer ces données en mémoire de masse. OCaml ne fournit pas de mécanisme de verrouillage mémoire analogue à la fonction `mlock` du langage C qui permet d'empêcher ces duplications au niveau système. Il est toutefois possible d'utiliser `mlock` par des appels externes (voir la fiche 18 concernant l'interfaçage avec du code externe).

Pour qu'elles soient effectives, les utilisations de `mlock` ne doivent pas porter sur les espaces mémoire alloués par le GC mais uniquement sur les espaces mémoire alloués par le code externe. Par exemple, les données construites à l'aide de la bibliothèque `Bigarray` sont ignorées par le GC et peuvent être protégées en utilisant `mlock`.

Références externes

– [ANA-SECU, 2011, §1.1.1] (Gestion des noms)
– [ANA-SECU, 2011, §1.1.4] (Mécanismes de gestion des expressions)
– [ANA-SECU, 2011, §1.2] (Traits impératifs)
– [MODE-EX, 2011, §1.2.4] (Comparaison et hachage)
– [MODE-EX, 2011, §1.5.3] (GC et données confidentielles)

Chapitre 3

Production de textes source

3.1 Styles de programmation

OCaml offre plusieurs styles de programmation : fonctionnel (fiche 10), impératif (fiche 11) et objet (fiche 5 de la section 2). Le style fonctionnel pur offre de nombreuses garanties mais toute application requiert un certain nombre d'effets de bord, au moins pour acquérir des données, communiquer des résultats, effacer de manière explicite une donnée en mémoire, etc.

Fiche 10 **Programmation fonctionnelle pure**

Un programme fonctionnel pur est une composition de fonctions (sans effet de bord) appliquées à ses entrées et à des résultats intermédiaires. Dans un tel programme, toute variable, toute expression a toujours la même valeur (transparence référentielle), ce qui facilite la relecture du programme, son analyse, son test et sa maintenance.

R-10-45 Choisir un style fonctionnel pur pour implanter les fonctionnalités de l'application, encapsuler les effets de bord nécessaires.

Cette recommandation est à compléter par celles des fiches portant sur les impacts des traits non fonctionnels (fiche 11 pour l'affectation et les effets de bord, fiche 15 pour les exceptions et fiche 5 pour les traits objet).

Si une donnée (éventuellement mutable) ne doit être utilisée que par une fonction (ou un groupe de fonctions), elle peut être définie localement à la

fonction. Elle sera alors encapsulée dans sa fermeture et ne sera donc accessible que par la fonction elle-même. Il est possible d'obtenir la même forme d'encapsulation par application partielle. Les fermetures peuvent donc être utilisées comme moyen d'encapsulation sûr. Ce moyen permet de répondre à des besoins ponctuels d'encapsulation simple.

R-10-46 Les données sensibles nécessaires à l'exécution d'une fonction peuvent être encapsulées dans sa fermeture.

Le style de programmation fonctionnel implique une grande utilisation de fonctions récursives (marquées par le mot-clé `rec`). Il est recommandé de vérifier les appels des fonctions récursives non terminales pour éviter tout débordement de pile.

R-10-47 Estimer le nombre d'appels récursifs non terminaux engendrés pour chaque utilisation d'une fonction récursive.

OCaml met en place le partage de sous-structures de données, s'il est explicité dans le texte source par une déclaration ou un filtrage. Cela facilite la manipulation de grosses structures de données sans avoir à utiliser la mutabilité.

R-10-48 Si la taille des données manipulées le nécessite, favoriser le partage de sous-structures.

Le filtrage (voir la fiche 14) permet de manipuler des données structurées tout en gardant un style fonctionnel pur (aucune manipulation de pointeurs).

R-10-49 Privilégier le filtrage comme moyen de définition de fonctions.

Les entiers de type `int` sont bornés par les constantes `max_int` et `min_int` dont les valeurs différent selon le processeur.

L'arithmétique sur les entiers (types `int`, `int32`, `int64`, `nativeint`*) est modulaire.*

R-10-50 Vérifier le non débordement d'entiers des opérations arithmétiques. Utiliser une bibliothèque de grands entiers si nécessaire.

Références externes

- [ETAT-LANG, 2011, §1.1] (Noyau fonctionnel)
- [ETAT-LANG, 2011, §1.1.3] (Construction de fonctions)
- [ANA-SECU, 2011, §1.1.2] (Étude de la fonctionnalité)
- [Leroy *et al.*, 2010, §1.3] (*Functions as values*)

Fiche 11 **Affectations et effets de bord**

OCaml offre des traits impératifs classiques, qui sont l'affectation de données mutables, les boucles, les effets de bord et les exceptions (voir la fiche 15). Il n'y a, en revanche, pas d'arithmétique de pointeurs en OCaml.

Comme indiqué par la fiche 10, il est préférable de favoriser le style fonctionnel de programmation pour faciliter la relecture et la compréhension du programme. Cela n'implique pas de refuser toute utilisation de traits impératifs. Les communications avec l'environnement extérieur au système se font par effet de bord, la levée et le rattrapage d'exceptions permettent de traiter les cas d'erreur de manière très claire, la neutralisation explicite du contenu d'une zone mémoire se fait par affectation de cette zone.

Cependant, l'utilisation de valeurs mutables est dangereuse à plusieurs titres. La perte de la transparence référentielle complique la relecture, la vérification et le test, qui doivent être faits en prenant en compte l'évolution de ces valeurs au cours de l'exécution. De plus, toute valeur mutable peut être modifiée accidentellement ou indûment. L'encapsulation permet de protéger les variables mutables de modification indues. Mais celle-ci peut être contournée (voir la fiche 8).

R-11-51 Justifier l'utilisation de valeurs mutables. Les protéger par un des mécanismes d'encapsulation recommandés (voir la fiche 8).

Structures de données avec sous-structures mutables

Il existe plusieurs familles de types pouvant comporter des sous-structures mutables : les références (`ref`), les enregistrements à champs mutables, les types objet dont certains champs peuvent être mutables, les tableaux et les chaînes de caractères ainsi que tous les types construits à partir des types venant d'être mentionnés.

R-11-52 Adapter strictement la portée des variables mutables au besoin de l'application, en la justifiant. Vérifier que les variables mutables ne sont pas indûment partagées par un mécanisme d'*aliasing*.

Il est possible d'interdire l'utilisation des références dans un texte source, en définissant un *parser* adapté avec l'outil `camlp4`. Si justifié, il est possible de définir un type (à base de sommes et d'enregistrements) approprié aux structures devant comporter des parties mutables afin de pouvoir utiliser le filtrage sur ce type et de mieux contrôler les manipulations de ses valeurs.

R-11-53 Adapter strictement l'utilisation de champs mutables dans les enregistrements et les objets aux besoins de l'application, en la justifiant.

Effets de bord

Les effets de bord peuvent complexifier le comportement d'un programme et le rendre plus imprévisible. Pour garder la maîtrise du comportement du programme, il est préférable de bien encadrer les effets de bord en les localisant dans des modules dédiés.

R-11-54 Encapsuler les effets de bord. (analogue à **R-9-44**)

R-11-55 Identifier les utilisations des fonctions de la bibliothèque standard qui modifient l'environnement global d'exécution (cf. table 3.1).

Chaînes de caractères

Les chaînes de caractères (type `string`) sont des tableaux mutables de caractères. Elles servent à deux utilisations distinctes : la manipulation de suites de caractères destinées à l'impression et le calcul de bas niveau sur des vecteurs mutables d'octets (*buffer* d'entrées–sorties, etc.). La mutabilité peut compromettre l'intégrité des données dans les deux cas. Elle n'est cependant pas nécessaire à la première utilisation. Il est donc préférable de séparer ces deux utilisations dans la conception du programme. Comme OCaml ne sépare pas les deux usages, il est souhaitable de définir, pour les besoins de la première utilisation, un module encapsulant les opérations sur les chaînes et exportant un type abstrait non mutable.

L'exemple suivant donne un exemple d'un tel module qui encapsule certaines opérations sur les chaînes de caractères.

```
module String_non_mutables : sig
  type t
  val of_string : string -> t
  val to_string : t -> string
  val get : t -> int -> char
  val sub : t -> int -> int -> t
  val concat : t -> t list -> t
  val (^^^) : t -> t -> t
end = struct
```

```
10    type t = string
11    let of_string = String.copy
12    let to_string = String.copy
13    let get = String.get
14    let sub = String.sub
15    let concat = String.concat
16    let (^^^) = (^)
17  end
```

R-11-56 Justifier la modification en place de chaînes de caractères.

Contrairement aux autres littéraux mutables (tableaux, enregistrements avec champs mutables), les chaînes de caractères littérales ne sont pas réallouées à chaque évaluation (voir le paragraphe de [MODE-EX, 2011, §1.5.1] portant sur le partage). La modification d'un littéral change toutes les occurrences de ce littéral dans le texte source. La fonction `String.copy` permet d'éviter ce problème en forçant la copie.

L'exemple suivant illustre la différence entre une chaîne de caractères littérale et un littéral tableau ainsi que l'utilisation de `String.copy` pour forcer la copie de la chaîne littérale.

```
1  let gen_t () = [| 1; 2; 3; 4 |];;
2  val gen_t : unit -> int array = <fun>
3  let t = gen_t ();;
4  val t : int array = [|1; 2; 3; 4|]
5  t.(0) <- 35; t;;
6  - : int array = [|35; 2; 3; 4|]
7  gen_t ();;
8  - : int array = [|1; 2; 3; 4|]
9  let gen_s1 () = "abcd";;
10 val gen_s1 : unit -> string = <fun>
11 let s1 = gen_s1 ();;
12 val s1 : string = "abcd"
13 s1.[0] <- 'X'; s1;;
14 - : string = "Xbcd"
15 gen_s1 ();;
16 - : string = "Xbcd"
17 let gen_s2 () = String.copy "abcd";;
18 val gen_s2 : unit -> string = <fun>
19 let s2 = gen_s2 ();;
```

```
20 val s2 : string = "abcd"
21 s2.[0] <- 'X'; s2;;
22 - : string = "Xbcd"
23 gen_s2 ();;
24 - : string = "abcd"
```

R-11-57 N'utiliser que des copies de chaînes de caractères littérales obtenues avec `String.copy`.

R-11-58 Encapsuler les chaînes de caractères dont on souhaite préserver l'intégrité dans un type abstrait (voir la fiche 8) pour garantir leur non mutabilité.

Chaînes de caractères littérales de la bibliothèque standard

Il est à noter que la bibliothèque standard fait usage de chaînes de caractères littérales qu'il convient de manipuler avec précaution car leurs valeurs peuvent être modifiées lors de l'exécution du programme. La table 3.2 donne la liste de toutes les fonctions de la bibliothèque standard retournant des chaînes de caractères littérales. La table 3.3 donne la liste des utilisations dans la bibliothèque standard de chaînes de caractères littérales en argument d'exceptions.

R-11-144 Contrôler les manipulations de chaînes de caractères littérales de la bibliothèque standard (cf. tables 3.2 et 3.3) en cas de leur utilisation.

L'exemple suivant illustre la modification de la chaîne de caractère littérale `"false"` de la bibliothèque standard.

```
1 let s = string_of_bool false;;
2 val s : string = "false"
3 s.[0] <- 't';
4 s.[1] <- 'r';
5 s.[2] <- 'u';
6 s.[3] <- 'e';
7 s.[4] <- ' ';;
8 - : unit = ()
9 string_of_bool false;;
10 - : string = "true "
```

Références externes

- [ETAT-LANG, 2011, §1.2.1] (Langages fonctionnels et effets de bord)
- [ETAT-LANG, 2011, §1.2.2] (Traits impératifs)
- [ANA-SECU, 2011, §1.2] (Traits impératifs)
- [Leroy *et al.*, 2010, §1.5] (*Imperative features*)

Modules	Fonctions	Commentaires
`Pervasives`	`at_exit`	Fonction monotone ne faisant qu'augmenter la liste des actions à effectuer à la clôture du programme.
`Callback`	`register` `register_exception`	Fonction monotone ne faisant qu'ajouter des valeurs de rappel pour l'interfaçage avec C.
`Format`	`set_*`	Fonctions de configuration du *pretty-printer* par défaut.
	`pp_set_*`	Fonctions de configuration d'un *pretty-printer*.
`Gc`	`set`	Fonction de paramétrage du GC.
`Parsing`	`*`	Fonctions de parsing appelées par `ocamlyacc`.
`Printexc`	`record_backtrace`	Fonction d'activation de l'enregistrement de la pile de levée d'exception : ne changeant pas le comportement du programme.
`Random`	`*`	Fonctions du générateur de nombres pseudo-aléatoires.
`Sys`	`argv`	Arguments de la ligne de commande représentés par un tableau de chaîne de caractères : ces valeurs sont donc mutables.

	`executable_name`
	`interactive`
	`os_type`
		Chaînes de caractères représentant le contexte d'exécution : ces valeurs sont donc mutables.

	`signal` `set_signal`	Fonctions de rajout ou de remplacement de gestionnaires de signaux.

TABLE 3.1 – Fonctions de la bibliothèque standard modifiant l'environnement global

Ce tableau recense les fonctions positionnant des variables globales ou modifiant l'état du système. Il n'inclut pas les fonctions effectuant des entrées–sorties.

Modules	Fonctions	Commentaires
`Pervasives`	`string_of_bool`	Fonction retournant les littéraux `"true"` et `"false"`.
`Char`	`escaped`	Fonction retournant un littéral pour les caractères : `'`, `\`, `\n`, `\t`, `\r` et `\b`.

`Filename`	`current_dir_name`
	`parent_dir_name`
	`dirname`
	`dir_sep`
	`temp_dir_name`
		Fonctions pouvant retourner les littéraux représentant les répertoires courant (`"."`), parent (`".."`) et temporaire ainsi que le séparateur de répertoires (`"/"`).

`Printexc`	`to_string`	Fonction retournant des littéraux pour les exceptions `Out_of_memory` et `Stack_overflow`.
	`get_backtrace`	Fonction retournant un littéral si le mode débogue de compilation n'est pas utilisé.

`Sys`	`argv`
	`executable_name`
	`os_type`
	`ocaml_version`
		Fonctions retournant des littéraux définis au lancement du programme.

TABLE 3.2 – Fonctions de la bibliothèque standard retournant des chaînes de caractères littérales

Exceptions	Utilisations avec littéral en argument	Commentaires
`Invalid_argument`	`Arg.parse_argv`	Rattrapage de l'exception avec filtrage du littéral `"bool_of_string"`.
	appels à `invalid_arg` levées de l'exception	Levées de l'exception avec un littéral en argument.
`Failure`	`Arg.parse_argv`	Rattrapage de l'exception avec filtrage des littéraux `"int_of_string"` et `"float_of_string"`.
	appels à `failwith`	Levées de l'exception avec un littéral en argument.

`Match_failure`
`Sys_error`
`Arg.Help`
`Arg.Bad`
`Scanf.Scan_failure`
`Stream.Error`
	aucune	Pas de risque d'altération car la bibliothèque standard utilise ces exceptions sans chaîne de caractères littérale.

TABLE 3.3 – Exceptions prenant en argument des chaînes de caractères littérales et leurs utilisations dans la bibliothèque standard

Fiche 12 Portée des identificateurs

La portée statique de OCaml et son orientation fonctionnelle (voir la fiche 10) facilitent la compréhension d'un programme par analyse de la portée de ses identificateurs.

R-12-59 Choisir la portée des identificateurs de manière à ne jamais laisser visible un identificateur devenu inutile.

Le masquage consiste à introduire une nouvelle déclaration d'un identificateur déjà déclaré. Il est autorisé en OCaml mais peut introduire des ambiguïtés et entraver la compréhension d'un programme.

R-12-60 Interdire tout masquage global.

R-12-61 Justifier tout masquage local.

Il est possible de faire référence à un identificateur mentionné dans l'interface d'un module en utilisant son nom qualifié ou en utilisant seulement son nom, après avoir importé le module complet, avec la directive `open`. Cette deuxième solution rend plus complexe la relecture de textes source et peut potentiellement masquer des identificateurs préalablement introduits.

R-12-62 Éviter d'utiliser la directive `open` ainsi que la syntaxe factorisant la qualification d'identifiants (i.e. `M.(x + y)`).

Le masquage peut avoir une incidence sur le comportement du programme en cas de masquage de fonctions, opérateurs ou éléments des bibliothèques standard.

R-12-63 Ne jamais masquer les définitions du module `Pervasives`.

R-12-64 Ne jamais masquer les noms des modules de la bibliothèque standard. (analogue à **R-1-6**)

R-12-65 Choisir des noms de modules significatifs et distincts. (analogue à **R-1-5**)

Références externes

– [ANA-SECU, 2011, §1.1.1] (Gestion des noms)
– [MODE-EX, 2011, §1.5.3] (GC et données confidentielles)

Fiche 13 **Représentation de données**

Le langage de types de OCaml fournit des outils puissants pour la représentation des données : types polymorphes, enregistrements, tableaux, types somme avec leurs constructeurs de valeur, etc. Un certain nombre de recommandations générales ont déjà été faites dans la fiche 6. L'égalité entre deux valeurs d'un même type est définie structurellement, sauf pour les types objet et les types fonctionnels.

R-13-66 Ne pas utiliser les classes pour définir des structures de données.

La bibliothèque standard contient des modules définissant les structures de données les plus couramment utilisées en algorithmique (`List`, `Stack`, `Queue`, `Set`, etc.) avec des traitements génériques efficaces.

R-13-67 Utiliser les types prédéfinis dans la bibliothèque standard pour représenter les structures de données utilisées en algorithmique.

Types concrets

Les données manipulées par l'utilisateur peuvent avoir des formes variées, qui ne peuvent pas toujours être directement décrites par les types prédéfinis dans la bibliothèque standard. Il existe de nombreuses recettes pour représenter toutes sortes de données par des tableaux, des listes chaînées, etc. Plutôt que d'utiliser des codages *ad hoc* pour se ramener à ces types prédéfinis, il est préférable de définir les nouveaux types reflétant le mieux possible les spécifications des données manipulées. De plus, les types définis par l'utilisateur sont tous différents, ce qui renforce le contrôle de la cohérence du texte source par le typage (voir la fiche 6).

R-13-68 Définir la représentation des données manipulées par des types reflétant directement leur spécification.

Le choix des étiquettes des enregistrements peut faciliter la lecture du texte source, en apportant une forme de documentation.

R-13-69 Préférer les types enregistrement plutôt que les types produit ou les classes pour représenter des n-uplets de données.

De même, le choix des constructeurs des valeurs de type somme peut aussi fournir une forme de documentation.

R-13-70 Représenter les données linéaires ou arborescentes par des types somme.

Il est possible que le compilateur infère un type plus général que celui attendu par le programmeur. Cela peut être le symptôme d'une erreur de programmation (fonction qui ignore tout ou partie d'un argument, etc.). Des éditeurs (voir [OUTILS-OCAML, 2011]) peuvent présenter à l'utilisateur les informations inférées par le compilateur (types, portée des identificateurs, nature des appels) via les options `-annot` et `-i`.

R-13-71 S'assurer que le type inféré correspond au type attendu.

Les types variants polymorphes (`[`Variant1; `Variant2]`) augmentent l'expressivité du langage mais rendent plus difficile la relecture de textes source.

R-13-72 Ne pas utiliser les types variants polymorphes.

Types abstraits

Dès qu'un type a été abstrait dans une interface de module, ses valeurs ne sont manipulables à l'extérieur du module que par les fonctions exportées par ce module. Cette encapsulation permet de contrôler la création et la manipulation de valeurs de ce type et de maintenir des invariants sur la représentation des données (voir la fiche 8).

R-13-73 Encapsuler les représentations de données devant satisfaire des invariants de représentation dans un module muni de l'interface appropriée.

Références externes

- [ETAT-LANG, 2011, §1.1.2] (Types et construction de données)
- [ANA-SECU, 2011, §1.1.3] (Étude du typage)
- [Leroy *et al.*, 2010, §6.8] (*Type and exception definitions*)

Fiche 14 **Filtrage**

Les traitements effectués par filtrage bénéficient d'une analyse associée. Les fonctions peuvent être définies par filtrage sur les types produit, enregistrement et somme avec vérification de l'exhaustivité et sur les types atomiques comme `int`, `char` par utilisation de littéraux.

R-14-74 Favoriser la représentation de données par des types structurés pour bénéficier du filtrage.

L'analyse d'exhaustivité du filtrage garantit que les traitements définis par filtrage n'oublient aucun cas. Cette analyse produit par défaut des avertissements de compilation plutôt que des erreurs. Il est conseillé d'imposer l'exhaustivité des filtrages. Ceci peut être fait en suivant la recommandation **R-21-114**.

R-14-75 N'utiliser que des filtrages exhaustifs.

L'analyse de filtrage identifie les filtres non utilisés. Ils témoignent d'une erreur de conception de la définition par filtrage. Comme précédemment il est possible de transformer les avertissements en erreurs. Ceci peut être fait en suivant la recommandation **R-21-114**.

R-14-76 Ne jamais laisser des filtres non utilisés dans une définition par filtrage.

L'exhaustivité peut être obtenue en décrivant par des filtres les différentes formes des valeurs d'un type donné. Il est possible également d'utiliser un filtre *attrape-tout* qui filtre tous les cas non décrits par les filtres introduits avant lui. Les filtrages utilisant un attrape-tout sont dits fragiles. Les filtrages fragiles ont l'intérêt d'alléger l'écriture et donc de faciliter la relecture. Cependant, si une version ultérieure du logiciel étend le type avec un constructeur supplémentaire, la version antérieure de la fonction définie par filtrage pourra encore être compilée sans avertissement. Mais les cas supplémentaires seront traités par l'attrape-tout. Cela est une source d'erreurs connue. Pour se prémunir contre cette erreur, il suffit d'activer la vérification de ces filtrages fragiles en utilisant l'option `-w +4`.

R-14-77 Détecter les filtrages fragiles avant toute modification de la définition d'un type, en recompilant avec l'option `-w +4`.

Les gardes permettent d'ajouter à un filtre une condition évaluée dynamiquement. Le membre droit associé au filtrage d'une valeur n'est alors exécuté que si la garde est satisfaite. Sinon, le filtrage se poursuit avec le filtre suivant. L'utilisation de ces gardes complique la relecture de code et rend la localisation d'erreurs plus difficile puisque leur survenue peut dépendre de l'exécution.

R-14-78 Justifier l'utilisation de gardes.

La satisfaction d'une garde dépendant de l'exécution, la réalisation d'un effet de bord inclus dans une garde est imprévisible.

R-14-79 Ne pas mettre d'effet de bord dans les gardes.

Références externes

– [ANA-SECU, 2011, §1.1.5] (Filtrage)
– [Leroy *et al.*, 2010, §6.6] (*Patterns*)

Fiche 15 **Exceptions**

Le mécanisme de levée et rattrapage d'exception est une structure de contrôle non locale car le `raise e` et le `try ... with ...` rattrapant `e` peuvent se situer dans des parties du code apparemment non connectées (surtout avec la forme *attrape-tout*). Cette structure crée ainsi des chemins d'exécution difficilement identifiables dans un code, d'autant plus que le typage ne mentionne pas la présence d'exceptions : les exceptions ont pour type `exn`, mais une fonction de type `int -> int` peut ou non lever une exception. Contrairement à d'autres langages, les informations concernant les exceptions qu'une fonction lève ou rattrape ne sont pas fournies par le compilateur.

R-15-80 Rattraper les exceptions avec un filtrage nominatif des exceptions plutôt que l'*attrape-tout*, sauf dans la fonction principale ou en cas de finalisation.

Une levée d'exception jamais rattrapée se termine par un message sur la console. Sachant qu'une exception peut transporter une valeur, il est nécessaire d'empêcher ces affichages.

R-15-81 N'utiliser l'*attrape-tout* que localement pour finaliser un traitement (en re-levant l'exception rattrapée `try ... with e -> begin ...; raise e end`) sauf dans la fonction principale.

R-15-82 Utiliser un *attrape-tout* dans la fonction principale du programme pour récupérer toutes les exceptions qui pourraient être levées sans être rattrapées.

Les exceptions peuvent être paramétrées. Ces paramètres sont définis par le programmeur et sont les seules informations sur l'environnement de la levée d'exception qui seront accessibles au point du programme où le rattrapage de l'exception est fait. Les exceptions ne divulguent donc pas d'autres informations que celles que le programmeur a choisi de divulguer.

R-15-83 Interdire le passage de données sensibles en paramètre d'exception, même si elles sont encapsulées (analogue à **R-9-40**)

R-15-84 Vérifier que les bibliothèques utilisées ne font aucun passage de données sensibles en paramètre d'exception. (analogue à **R-9-41**)

R-15-85 Interdire le masquage des noms d'exception.

Références externes

– [ANA-SECU, 2011, §1.2.3] (Exceptions)
– [Leroy *et al.*, 2010, §6.7] (*Expressions*)

3.2 Sécurité et constructions prohibées

OCaml est un langage statiquement et fortement typé. Le typage assure un contrôle strict non seulement de la conformité de l'utilisation des valeurs avec leur type mais également de la conformité de l'utilisation des identificateurs avec leur portée (voir la fiche 12), donc une certaine forme de cloisonnement. Cependant, certaines constructions du langage peuvent briser la protection apportée par le typage et sont dites non sûres.

Fiche 16 **Modules non sûrs**

Module `Obj`

Les opérations du module `Obj` permettent d'attribuer un type quelconque à une valeur quelconque brisant ainsi la protection apportée par le typage, le programme étant quand même accepté par le compilateur. Il est rappelé que le programmeur en OCaml ne manipule jamais explicitement de pointeurs et que la compilation garantit que tous les accès à la mémoire exécutés sont corrects (il ne peut pas y avoir de débordement de tableaux ni de pointeur invalide). Cette garantie est invalidée par l'utilisation du module `Obj`. Les fonctions du module `Obj` permettent aussi d'accéder à des variables d'instance d'un objet et à ses méthodes privées cachées en forçant son type.

R-16-86 Proscrire l'utilisation du module `Obj`.

Module `Marshal`

Le module `Marshal` propose des fonctions de sérialisation et désérialisation pour la communication ou l'enregistrement de valeurs OCaml. Une valeur est sérialisée (fonctions `Marshal.to_*`) sous forme binaire pour être enregistrée dans un fichier ou transmise sur un canal de communication. Seule sa valeur est enregistrée, son type ne l'est pas. Une valeur sérialisée ne portant pas son propre type, celui-ci doit être fourni explicitement par le développeur au moment de la désérialisation (par annotation de type). La désérialisation (fonctions `Marshal.from_*`) se fait donc sans vérification de type. Cette construction peut donc briser la protection apportée par le typage. De plus, la sérialisation, qui ne brise pas le typage, permet de contourner l'encapsulation par type abstrait et même, avec l'option `Closures`, l'encapsulation par une fermeture.

Il est à noter que les données sérialisées par le module `Marshal` sont sous forme binaire mais ne sont pas chiffrées, elles restent donc facilement lisibles.

R-16-87 Proscrire l'utilisation du module `Marshal`

Il est conseillé de définir un format de représentation des données pour le stockage ou la communication et les *parser–printer* associés. L'écriture du *parser* en OCaml permet de retrouver les garanties apportées par le typage sur les valeurs entrantes.

R-16-88 Choisir un format de communication de données permettant différentes formes de contrôle.

La distribution OCaml propose deux générateurs de parsers : **`ocamlyacc`** et **`camlp4`**. Elle inclut aussi des bibliothèques de manipulation typée d'entrées–sorties (modules `Printf`, `Scanf`, `Format` et `Stream`).

R-16-89 Utiliser les outils de la distribution OCaml pour définir un *printer* et un *parser* adaptés au format de communication de données choisi.

Module `Printexc`

Le mécanisme d'exceptions (voir la fiche 15) rend possible le contournement de l'encapsulation des types abstraits, par utilisation du module `Printexc`. Celui-ci permet dans certains cas d'examiner les données passées à une exception sans tenir compte du type de ces données. Les fonctions du module `Printexc` sont donc non sûres. De plus, une exception non rattrapée provoque l'affichage partiel de l'exception et de ses arguments sur la console (quel que soit le mode d'exécution). Les fonctions de ce module ne peuvent donc être utilisées que dans la phase de mise au point du code et doivent être éliminées dans le code mis en exploitation.

R-16-90 Proscrire l'utilisation du module `Printexc`.

Références externes

- [ANA-SECU, 2011, §1.1.3] (Étude du typage)
- [MODE-EX, 2011, §1.2.5] (Constructions invalidant l'analyse statique)
- [MODE-EX, 2011, §1.3.5] (Interfaçage avec du code en C)
- [MODE-EX, 2011, §2.2.3] (Interfaçage avec du code C)
- [MODE-EX, 2011, §3.2.3] (Interfaçage avec du code C)
- [Leroy *et al.*, 2010, Ch. 18] (*Interfacing C with OCaml*)

Fiche 17 **Fonctions non sûres**

L'accès et la modification des chaînes de caractères et des tableaux est contrôlée dynamiquement par OCaml à l'aide d'une instrumentation du code compilé empêchant les accès non contrôlés à la mémoire. Cette vérification de non dépassement des bornes peut toutefois être désactivée par l'option `-unsafe` du compilateur ou par l'utilisation de fonctions de la bibliothèque standard de préfixe `unsafe_`.

R-17-91 Ne pas utiliser l'option `-unsafe`. (analogue à **R-21-115**)

Les fonctions `unsafe_*` de la bibliothèque standard sont des versions de fonctions d'accès et de modification de chaînes de caractères et de tableaux qui désactivent les vérifications. Elles se trouvent dans les modules :
- `Pervasives` (importé par défaut),
- `Array`,
- `ArrayLabels`,
- `Bigarray`,
- `Char`,
- `Printf.CamlinternalPr.Sformat`,
- `StdLabels`,
- `String`,
- `StringLabels`.

R-17-92 Ne pas utiliser les fonctions `unsafe_*`.

La syntaxe du langage OCaml ne permet aucune manipulation d'une variable avant son initialisation, la seule exception étant les chaînes de caractères introduites par la fonction `String.create`. Celle-ci effectue une allocation mémoire sans initialisation, permettant ainsi de consulter des portions de mémoire précédemment allouées.

R-17-93 Proscrire l'utilisation de `String.create` et la remplacer par `String.make`. Vérifier également que les bibliothèques utilisées n'effectuent pas d'appels à `String.create`.

Randomisation des tables de hachage

Les versions de OCaml jusqu'à 3.12.1[1] *ne comportent qu'une fonction de hachage simple et non randomisable. Ceci peut conduire*

1. L'étude porte sur la version 3.12.0 de OCaml.

à des attaques de déni de service sur les programmes qui utilisent des tables de hachage.

Cette vulnérabilité a pour identifiant CVE-2012-0839, elle est décrite à l'adresse suivante :
`http://nvd.nist.gov/nvd.cfm?cvename=CVE-2012-0839`

Le problème de non randomisation des tables de hachage est résolu à partir de la version 4.00.0 de OCaml par l'ajout d'une option de randomisation des tables de hachage.

R-17-147 À partir de la version 4.00.0 de OCaml, utiliser systématiquement l'option de randomisation des tables de hachage.

Dans les versions antérieures de OCaml (jusqu'à la version 3.12.1), l'utilisation de `Hashtbl` peut être évitée en utilisant les fonctionnalités des modules `Set` et `Map` de la bibliothèque standard car ces structures de données sont naturellement résistantes aux collisions.

R-17-148 Dans la version 3.12.1 de OCaml, utiliser les modules `Set` et `Map` à la place des tables de hachage.

Références externes

- [MODE-EX, 2011, §1.2.5] (Constructions invalidant l'analyse statique)
- [MODE-EX, 2011, §1.5.1] (Allocation, désallocation)
- Vulnérabilité CVE-2012-0839 décrite à l'adresse suivante : `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2012-0839`

Fiche 18 Construction `external` : interfaçage avec C

OCaml permet l'interfaçage avec du code natif. Ce code peut être obtenu par compilation de n'importe quel langage mais est le plus souvent issu d'un source en C. La construction `external` permet de déclarer une fonction dont l'appel lancera l'exécution d'une fonction C donnée. Par exemple, `external fn_ocaml : int -> int = "fn_c"` déclare une fonction `fn_ocaml` de type OCaml `int -> int`. L'appel de `fn_ocaml` exécutera la fonction `fn_c` en lui passant son paramètre effectif. Le type de la fonction est donné explicitement par le programmeur.

Les accès mémoire par le code C appelé dans un programme OCaml se font dans l'espace mémoire de ce programme OCaml. Ce code C peut donc lire et écrire dans cet espace et donc consulter et modifier arbitrairement les données du programme OCaml. Cette construction `external` est donc non sûre. Cependant, il peut être nécessaire de l'utiliser pour réaliser des actions de bas niveau (voir fiche 9) ou pour réutiliser du code pré-existant écrit dans un autre langage que OCaml (par exemple, bibliothèques de chiffrement).

R-18-94 Ne pas utiliser de code externe, sauf pour des raisons dûment justifiées et documentées. (analogue à **R-4-14**)

R-18-95 Tout code externe utilisé doit être, si possible, recompilé à partir du son source.

R-18-96 Vérifier le code C utilisé par analyse du texte source. S'assurer de son intégrité au moyen par exemple d'une signature cryptographique.

Références externes

- [MODE-EX, 2011, §1.3.5] (Interfaçage avec du code en C)
- [MODE-EX, 2011, §2.2.3] (Interfaçage avec du code C)
- [MODE-EX, 2011, §3.2.3] (Interfaçage avec du code C)
- [Leroy *et al.*, 2010, Ch. 18] (*Interfacing C with OCaml*)

Fiche 19 Module `Dynlink` : chargement dynamique de code

OCaml fournit un mécanisme de chargement dynamique de code avec le module `Dynlink` de la bibliothèque standard rendant possible la spécialisation d'applications par chargement de *plug-ins* à l'exécution. Ce chargement, même s'il apporte des fonctionnalités intéressantes, est dangereux car les fonctionnalités réelles du code chargé ne peuvent pas être contrôlées. De plus OCaml n'effectue que très peu de vérifications sur le code chargé dynamiquement.

R-19-97 Éviter le chargement dynamique de code.

Le chargement dynamique sert souvent à spécialiser une application selon les interactions avec l'environnement extérieur (utilisateur, composant, etc.). Une première alternative peut être d'incorporer dans le texte source de l'application toutes les spécialisations autorisées. Une seconde alternative est de définir un langage d'interaction avec cet environnement, permettant de définir un traitement spécialisé par combinaison de traitements atomiques décrits dans le source de l'application. Si toutefois il n'est pas possible d'éviter un recours à `Dynlink`, les recommandations suivantes doivent impérativement être respectées.

R-19-98 Toute utilisation du module `Dynlink` doit être dûment justifiée et contrôlée.

Configuration de `Dynlink`

Le système de configuration de `Dynlink` permet de contrôler le chargement de codes *plug-ins* compilés, en restreignant la liste des modules que ces codes peuvent importer.

R-19-99 Contrôler avec `Dynlink.allow_only` les fonctions qu'un *plug-in* peut utiliser en n'autorisant que des fonctions sûres, regroupées dans un module dédié.

R-19-100 Proscrire les utilisations des fonctions `reset`, `add_interfaces` et `add_variable_units` de `Dynlink`.

En modèle d'exécution bytecode seulement, il est possible d'interdire les appels à des fonctions C (via la construction `external`).

R-19-101 En bytecode, interdire l'utilisation des primitives C (construction `external`) avec `Dynlink.allow_unsafe_modules false`.

L'appel `Dynlink.allow_unsafe_modules false` *est inopérant en code natif.*

L'exemple suivant de chargement dynamique n'autorise, pour les modules chargés dynamiquement, qu'un module `ModuleSur` à définir et interdit les appels externes (restriction uniquement valable en modèle bytecode). Ces contrôles doivent s'accompagner de la maîtrise de la compilation du fichier chargé dynamiquement (ici le fichier `a.cmo`).

```
1 Dynlink.allow_only ["ModuleSur"];;
2 Dynlink.allow_unsafe_modules false;;
3 Dynlink.loadfile "a.cmo";;
```

Vérifications à effectuer

La fonction `Dynlink.loadfile` permet de charger un *plug-in* sous la forme d'un code compilé natif ou d'un bytecode suivant le modèle d'exécution. Le nom du fichier du *plug-in* à charger est une chaîne de caractères. Ce nom est donc représenté par une valeur mutable, ce qui pourrait permettre un détournement.

R-19-102 Contrôler l'argument de `Dynlink.loadfile`.

Configurer `Dynlink` suivant les recommandations précédentes ne suffit pas. Le code chargé dynamiquement avec `Dynlink` est du code compilé, pour lequel OCaml ne fournit pas de mécanisme de vérification. Il est donc préférable de disposer des textes source des *plug-in*, de maîtriser leur compilation et de s'assurer de l'intégrité des fichiers compilés des *plug-ins*.

R-19-103 Effectuer une relecture des textes source des *plug-ins*.

R-19-104 Maîtriser la compilation des fichiers *plug-ins*.

R-19-105 Contrôler les droits d'accès aux fichiers *plug-ins*.

Références externes

- [MODE-EX, 2011, §1.3.4] (Chargement dynamique de code OCaml compilé)
- [MODE-EX, 2011, §2.2.2] (Chargement dynamique de code natif OCaml)
- [MODE-EX, 2011, §3.2.2] (Chargement dynamique de bytecode OCaml)
- [Leroy *et al.*, 2010, Ch. 27] (*The dynlink library : dynamic loading and linking of object files*)

Fiche 20 **Vérification de bibliothèques importées**

La bibliothèque standard de OCaml utilise des constructions non sûres utilisées de manière sûre. La fiche 24 porte sur l'installation de cette bibliothèque.

R-20-106 Faire l'installation du langage à partir d'une distribution de confiance. (analogue à **R-24-130**)

En dehors de la bibliothèque standard, des bibliothèques externes OCaml peuvent être utilisées. Les recommandations qui suivent peuvent aussi s'appliquer à certains modules de l'application dont la spécification oblige à violer certaines recommandations.

R-20-107 Procéder à la relecture des textes source dont on ne peut affirmer l'innocuité. (analogue à **R-7-33**)

Le module `Obj` peut être utilisé pour des opérations de bas niveau, nécessaires à l'exécution de l'application.

R-20-108 Contrôler par relecture du code les utilisations du module `Obj` dans les textes source des bibliothèques importées.

Si la spécification impose que l'échange de données soit fait par sérialisation/-désérialisation, il faut alors ajouter un mécanisme de protection.

R-20-109 Contrôler si possible le type de la fonction sérialisant la donnée.

R-20-110 Contrôler l'origine de la valeur à désérialiser en ajoutant par exemple une signature cryptographique à la donnée sérialisée.

R-20-111 Toute utilisation de `Obj` ou de `Marshal` doit être justifiée, contrôlée et, si possible, encapsulée.

R-20-112 Ne tolérer les utilisations des fonctions `unsafe_*` dans les textes source de bibliothèques externes que si elles sont dûment justifiées.

R-20-113 Choisir les options de compilation qui maximisent les vérifications. Voir la fiche 21 pour le détail de ces options.

Références externes

– [ANA-SECU, 2011, §1.1.3] (Étude du typage)
– [MODE-EX, 2011, §1.2.5] (Constructions invalidant l'analyse statique)

Chapitre 4

Compilation

Fiche 21 **Options de compilation**

Le compilateur OCaml effectue de nombreuses vérifications. Certaines sont activées par défaut mais peuvent être désactivées (vérification dynamique de non dépassement, vérification des assertions, la plupart des avertissements, etc), d'autres sont inactives par défaut et peuvent être activées (avertissements 4, 6, 7, 9, 27, 28 et 29 de la liste donnée par l'option `-warn-help`, types unit pour les séquences). Le plus haut niveau de vérifications est conseillé sauf celui du filtrage fragile (avertissement 4).

R-21-114 Activer l'ensemble des vérifications avec l'option `-w +a-4` et considérer comme une erreur de compilation tout avertissement avec l'option `-warn-error +a`.

R-21-115 Ne pas utiliser les options `-unsafe`, `-noassert`, `-rectypes`.
(analogue à **R-17-91**)

R-21-116 Utiliser l'option `-strict-sequence`.

Le compilateur permet d'enregistrer dans un fichier `.annot` les informations qu'il a inférées. Des environnements de programmation donnent accès à ces informations (voir [OUTILS-OCAML, 2011, §1.1]).

R-21-117 Utiliser l'option `-annot` en conjonction avec un éditeur pour vérifier les types inférés et la nature des appels récursifs (terminaux ou non).

L'option `-pp` permet de faire appel à un préprocesseur. OCaml est distribué avec le préprocesseur `camlp4` (voir la fiche 26).

R-21-118 Vérifier les commandes passées en argument à l'option `-pp`.

Certains modes de compilation favorisent le chargement statique ce qui diminue le risque de chargement de code malveillant au lancement du programme.

R-21-119 Préférer la production d'un exécutable complet en utilisant les modes de compilation avec chargement statique de code : compilation de code natif, ou compilation de bytecode avec les options `-custom` ou `-output-obj`.

Certaines options permettent de déterminer manuellement les fichiers à charger et les chemins des répertoires les contenant, offrant ainsi un meilleur contrôle sur l'exécutable produit.

R-21-120 Désactiver la liaison automatique des fichiers externes mentionnés par les bibliothèques avec l'option `-noautolink`.

R-21-121 Contrôler les arguments des options `-I`, `-dllib` et `-dllpath`. Justifier les utilisations de l'option `-linkall`.

Le compilateur OCaml est fourni avec un débogueur et un *profiler* dédiés. En phase de mise au point, on peut activer les modes débogue (option `-g`) et profilage (option `-p`) de compilation. En revanche, ces modes doivent être désactivés en phase d'exploitation.

R-21-122 En phase d'exploitation, il est impératif de désactiver les modes débogue et profilage de compilation.

(analogue à **R-3-10**, **R-23-128** et **R-25-137**)

La table 4.1 récapitule les recommandations d'usage des options de compilation données dans cette fiche.

Références externes

– [MODE-EX, 2011, §1.2.7] (Options de compilation)
– [MODE-EX, 2011, §2.1.1] (Options de compilation)
– [MODE-EX, 2011, §3.1.1] (Options de compilation)
– [MODE-EX, 2011, §5.4.2] (Options de compilation et de la boucle interactive)

Commande		Option	Phase	
`ocamlopt`	`ocamlc`		Mise au point	Exploitation
●	●	`-annot`	✚	
○	●	`-custom`	✚	✚
○	●	`-dllib <lib>`	✶	✶
○	●	`-dllpath <dir>`	✶	✶
●	●	`-g`		━
●	●	`-I <dir>`	✶	✶
●	●	`-linkall`	✶	✶
●	●	`-noassert`	━	━
●	●	`-noautolink`	✚	✚
●	●	`-output-obj`		✚
●	○	`-p`		━
●	●	`-pp <command>`	✶	✶
●	●	`-rectypes`	━	━
●	●	`-strict-sequence`	✚	✚
●	●	`-unsafe`	━	━
●	●	`-w +a-4`	✚	✚
●	●	`-warn-error +a`	✚	✚

TABLE 4.1 – Recommandations d'usage des options de compilation

Les recommandations d'usage (positive ✚, négative ━, et avertissement ✶) des options des commandes `ocamlopt`, `ocamlc` sont récapitulées dans ce tableau suivant pour les phases de mise au point et d'exploitation.

Fiche 22 **Processus de compilation**

Préprocesseur

Le compilateur peut appeler un pré-processeur pour analyser des fichiers `.ml`. Il est possible en particulier d'utiliser le pré-processeur **`camlp4`** pour valider certaines recommandations de sécurité. On peut par exemple écrire à l'aide de ce pré-processeur un *parser* qui n'acceptera aucune occurrence de **`ref`** dans le texte source ou qui n'acceptera aucune occurrence d'appel au module `Obj`.

L'exemple de texte source **`camlp4`** définit un préprocesseur interdisant les appels au module `Obj`. Le fichier de ce préprocesseur `pa_no_obj.ml` se compile avec les options suivantes : `ocamlc -c -I +camlp4 -pp "camlp4rf" pa_no_obj.ml`. Le préprocesseur obtenu peut ensuite être appelé lors de la compilation d'un fichier OCaml : `ocamlc -c -pp "camlp4o pa_no_obj.cmo" fichier.ml`. Une exception est levée au parsing si `fichier.ml` contient un appel au module `Obj`.

```
1  open Camlp4.PreCast; open Syntax;
2  exception Use_of_Obj;
3  DELETE_RULE Gram a_UIDENT: `UIDENT s END;
4  DELETE_RULE Gram a_UIDENT: `ANTIQUOT (""|"uid") s END;
5  EXTEND Gram
6    GLOBAL: a_UIDENT;
7    a_UIDENT:
8        [ [ `UIDENT "Obj" -> raise Use_of_Obj
9          | `UIDENT s ->  s ] ];
10 END;
```

R-22-123 Vérifier les textes source produits par `camlp4` en utilisant son *printer* de textes source OCaml.

Utilisation de bibliothèques pré-existantes

Lors de la production d'un exécutable les fichiers objets (natif ou bytecode selon le modèle) sont liés. Ces fichiers peuvent être compilés séparément et provenir d'une compilation extérieure. Effectuer la liaison d'un fichier correspond à accepter d'exécuter son code. Or les codes objet OCaml ne contiennent

pas les informations de typage des textes source. On ne peut donc pas se reposer sur le mécanisme de typage pour leur accorder confiance. Il faut donc d'une part vérifier les textes source des bibliothèques utilisées (suivant les recommandations de ce document) et d'autre part, vérifier que leur processus de compilation (souvent décrit dans un `Makefile`) suit aussi les recommandations de ce document. Au démarrage d'un programme OCaml, les valeurs *toplevel* de chaque module sont exécutées même si la bibliothèque n'est pas utilisée. Il est donc important de s'assurer que ces évaluations au chargement du module sont sûres.

R-22-124 Vérifier tous les textes source des bibliothèques utilisées et contrôler leurs processus de compilation.

L'option `-linkall` *inclut toutes les unités de compilation de toutes les bibliothèques passées en ligne de commande, les points d'entrée de leurs modules seront donc aussi exécutés.*

Pour ne pas avoir à vérifier des textes source non utilisés, il est préférable de reconstruire une bibliothèque dédiée à l'application, à partir des bibliothèques pré-existantes en n'y incorporant que les modules qui sont effectivement appelés.

R-22-125 Reconstruire une bibliothèque dédiée à l'application, vérifier ses fichiers source, vérifier ou redéfinir son processus de compilation.

R-22-126 En cas d'utilisation de l'option `-linkall`, vérifier les sources de toutes les unités de compilation de toutes les bibliothèques passées en ligne de commande (sans se limiter aux unités réellement utilisées par le programme).

Références externes

- [MODE-EX, 2011, §1.2] (Phases communes de compilation)
- [MODE-EX, 2011, §2.1.3] (Compilation séparée, fichiers objet `.cmx`, `.cmxs` et `.cmxa`)
- [MODE-EX, 2011, §3.1.3] (Compilation séparée, fichiers objet `.cmo` et `.cma`)
- [Leroy *et al.*, 2010, §2.5] (*Modules and separate compilation*)

Chapitre 5

Exécution

Fiche 23 **Système *runtime***

OCaml possède deux modèles d'exécution largement compatibles (le comportement d'un programme sera de manière générale le même quel que soit le modèle d'exécution). Quelques différences existent toutefois.

Exécution bytecode en mode débogue

L'exécution en modèle bytecode peut être faite en mode débogue même si le bytecode n'est pas produit en mode de compilation debogue (voir recommandation **R-3-10**). Ce mode d'exécution peut être activé simplement en définissant la variable d'environnement système `CAML_DEBUG_SOCKET` avec une *socket* donnée. Ce mode ainsi activé permet d'observer des valeurs manipulées par le programme et donc de profiter d'une élévation de privilèges pour accéder à des données confidentielles. Ce mode permet aussi de rejouer tout ou partie du déroulement du programme.

R-23-127 N'utiliser le modèle d'exécution bytecode que si un contrôle de la variable d'environnement `CAML_DEBUG_SOCKET` est possible.

(analogue à **R-25-139**)

Ce contrôle ne peut pas être fait par le programme OCaml lui-même car le mode débogue est activé avant le démarrage du programme.

Backtrace

Un mode particulier d'exécution permet d'accéder à la pile de levée d'exception ou *backtrace*. Ce mode pouvant divulguer des informations sur le comportement du programme, il est conseillé de l'empêcher en exploitation en compilant le programme sans l'option `-g`.

R-23-128 En exploitation, ne pas utiliser l'option `-g` de compilation (native ou bytecode) pour ne pas permettre une exécution en mode d'enregistrement de *backtrace*.

(analogue à **R-3-10**, **R-21-122** et **R-25-137**)

Traitement des signaux

Le traitement des signaux s'effectue à l'aide des modules `Sys` et `Unix` de la bibliothèque standard.

Il est à noter qu'il n'y a pas de gestion des signaux pendant la durée de l'exécution d'un code externe C ainsi que, en mode natif, dans une boucle n'effectuant pas d'allocation mémoire.

La configuration des traitements des signaux s'effectue avec les fonctions `Sys.signal`, `Sys.set_signal` et `Unix.sigprocmask` qui positionnent des variables globales. Il faut donc s'assurer de l'absence de reconfiguration ultérieure. Cela doit être fait par examen du texte source.

R-23-129 Contrôler les configurations du gestionnaire de signaux.

Références externes

- [MODE-EX, 2011, §1.4] (Parties communes des exécutifs)
- [MODE-EX, 2011, §2.3] (Exécution et interaction avec le système)
- [MODE-EX, 2011, §3.3] (Exécution et interaction avec le système)
- [MODE-EX, 2011, §5.1] (Variables d'environnement)

Chapitre 6

Installation et configuration

Fiche 24 **Installation de OCaml**

Les fichiers d'installation du langage comprennent les compilateurs, qui manipulent le texte source des programmes, et la bibliothèque standard, dont le code est chargé dans les exécutables. La modification de ces fichiers permet d'injecter du code malveillant dans n'importe quel programme compilé avec ces outils.

R-24-130 Faire l'installation du langage à partir d'une distribution de confiance. (analogue à **R-20-106**)

R-24-131 Empêcher toute modification des fichiers du système OCaml après leur installation, en restreignant les droits en écriture sur ces fichiers.

Privilèges système

Les compilateurs `ocamlopt`, `ocamlc`, et le générateur de boucles interactives `ocamlmktop` ne sont pas conçus pour une utilisation en milieu hostile et peuvent se prêter à des attaques par élévation de privilèges si on leur donne des droits privilégiés. De plus, la machine virtuelle `ocamlrun` et la boucle interactive `ocaml` sont capables d'exécuter un bytecode ou un texte source arbitraire. Leur donner des droits privilégiés amène trivialement à une attaque par injection de code. Ces interpréteurs permettent d'exécuter du code sans avoir à fabriquer un exécutable.

R-24-132 Ne pas donner de droits privilégiés aux compilateurs (`ocamlopt` et `ocamlc`), au générateur de boucles interactives (`ocamlmktop`), à la machine virtuelle (`ocamlrun`) ou aux boucles interactives (`ocaml`).

Références externes

– [MODE-EX, 2011, §5.2] (Privilèges système)
– [OUTILS-OCAML, 2011]

Fiche 25 **Variables d'environnement**

Des variables d'environnement sont utilisées par OCaml pour certaines configurations.

Les variables `OCAMLLIB` et `CAMLLIB` sont utilisées par le compilateur pour connaître l'emplacement de la bibliothèque standard lorsque celui-ci diffère de l'emplacement par défaut. En changeant la valeur de ces variables lors de la compilation, il est possible de substituer du code malveillant à la bibliothèque standard lors de la phase de chargement statique.

R-25-133 Vérifier les valeurs des variables `OCAMLLIB` et `CAMLLIB` avant la compilation : ces deux variables doivent être absentes de l'environnement.

Les variables `CAML_LD_LIBRARY_PATH`, `OCAMLLIB`, `CAMLLIB` sont utilisées à l'exécution des programmes bytecode pour déterminer l'emplacement des fichiers chargés au démarrage lorsque cet emplacement diffère de celui défini par défaut. Ces variables doivent *a priori* être absentes. En changeant la valeur de ces variables, il est possible de substituer du code malveillant à ces fichiers lors de la phase de chargement au démarrage.

R-25-134 En bytecode, vérifier l'absence ou contrôler les valeurs des variables `CAML_LD_LIBRARY_PATH`, `OCAMLLIB` et `CAMLLIB` avant l'exécution.

La variable `PATH` est utilisée par la machine virtuelle pour déterminer l'emplacement du fichier bytecode à exécuter (aussi bien avec l'option `-custom` qu'en mode normal). En changeant la valeur de cette variable, il est possible de substituer du code malveillant au programme tout entier. Si ce programme a des droits privilégiés, le code malveillant sera alors exécuté avec ces droits. En mode particularisé, cette attaque ne fonctionne pas sous le système Linux.

R-25-135 Ne pas donner de droits privilégiés à une programme compilé en modèle bytecode.

Les variables `OCAMLRUNPARAM` et `CAMLRUNPARAM` permettent d'activer le mode *backtrace* d'exécution pour les programmes compilés en mode débogue (option `-g`). Elles permettent aussi le mode de trace de *parsers* générés par `ocamlyacc` mais ne sont pas utilisées par les *parsers* `camlp4` et `menhir`[1], ce dernier étant un remplaçant compatible avec `ocamlyacc`.

1. `http://gallium.inria.fr/~fpottier/menhir/`

R-25-136 En phase d'exploitation, vérifier l'absence des caractères `b` et `p` dans la valeur des variables `OCAMLRUNPARAM` et `CAMLRUNPARAM` à l'exécution.

R-25-137 En phase d'exploitation, ne pas utiliser l'option `-g` de compilation.
(analogue à **R-3-10**, **R-21-122** et **R-23-128**)

R-25-138 En phase d'exploitation, ne pas utiliser de *parsers* produits par `ocamlyacc`. Compiler les *parsers* avec `menhir`.

La variable `CAML_DEBUG_SOCKET` est utilisée par la machine virtuelle pour communiquer avec un débogueur qui contrôle l'exécution et affiche les valeurs des variables. En changeant la valeur de cette variable, il est possible d'extraire les informations sensibles manipulées par le programme et de contrôler son déroulement dans une certaine mesure.

R-25-139 En bytecode, vérifier l'absence de la variable `CAML_DEBUG_SOCKET` avant l'exécution en phase d'exploitation.
(analogue à **R-23-127**)

Les compilateurs **`ocamlc`** et **`ocamlopt`** créent des fichiers temporaires dans un emplacement contrôlé par la variable `TMPDIR`. En changeant la valeur de cette variable, il est possible de forcer le compilateur à créer ses fichiers temporaires dans un répertoire non protégé et ensuite d'injecter du code malveillant en modifiant ces fichiers pendant la compilation.

R-25-140 Vérifier l'absence de la variable `TMPDIR` lors de la compilation.

Références externes

– [MODE-EX, 2011, §5.1] (Variables d'environnement)

Fiche 26 Configuration de `camlp4`

De même que les compilateur, l'outil **`camlp4`** n'est pas conçu pour une utilisation en milieu hostile, et lui donner des droits privilégiés peut conduire à une attaque par élévation de privilèges.

R-26-141 Ne pas donner de droits privilégiés au préprocesseur `camlp4`.

Les modules d'extension de **`camlp4`** peuvent changer le programme compilé entre la phase de *parsing* et les premières phases du compilateur, sans changement du texte source de ces programmes. Il faut donc vérifier que ces modules ne peuvent pas introduire de vulnérabilités dans les programmes à l'insu du programmeur.

R-26-142 Vérifier les textes sources des modules d'extension pour `camlp4` ainsi que leur processus de compilation.

R-26-143 Vérifier la liste des modules d'extension chargés par `camlp4` en utilisant son option `-loaded-modules`.

Références externes

– [MODE-EX, 2011, §1.2.1] (Préprocesseurs et **`camlp4`**)
– [OUTILS-OCAML, 2011]

Bibliographie

[ANA-SECU, 2011] Analyse des langages OCaml, Scala et F#. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L2.2.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC et Normation.

[ETAT-LANG, 2011] État des lieux des langages fonctionnels. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L2.1.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC et Normation.

[Leroy *et al.*, 2010] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. et Vouillon, J. (2010). The Objective Caml system release 3.12 – documentation and user's manual. INRIA. Disponible en ligne `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[MODE-EX, 2011] Modèles d'exécution du langage OCaml. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L3.1.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC et Normation.

[OUTILS-OCAML, 2011] Outils associés au langage OCaml. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L3.1.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC et Normation.

[TC2-RAP, 2012] Rapport d'évaluation. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec), seconde tranche conditionnelle (TC2) L7.2, ANSSI (2012). Étude menée par un consortium composé de SafeRiver et Oppida.

Table des tables

Acronymes

ANSSI Agence Nationale de la Sécurité des Systèmes d'Information
CCTP Cahier des Clauses Techniques Particulières
GC Garbage Collector (ramasse-miettes)
SGDSN Secrétariat Général de la Défense et de la Sécurité Nationale
TC1 Tranche Conditionnelle n° 1
TC2 Tranche Conditionnelle n° 2