Parsifal : utilisation de `camlp4` pour l'écriture rapide de *parsers*

Olivier Levillain[1,2]

1: Agence nationale de la sécurité des systèmes d'information (ANSSI)
51 boulevard Latour Maubourg, 75700 Paris 07 SP
olivier.levillain@ssi.gouv.fr
2: Télécom Sud Paris
9 rue Charles Fourier, 91011 Evry

Dans le cadre de ses activités d'expertise, le laboratoire sécurité des réseaux et protocoles (LRP) de l'agence nationale de la sécurité des systèmes d'information (ANSSI) est amené à étudier divers protocoles de communication. L'étude fine de ces protocoles passe par l'utilisation de *parsers*, ou dissecteurs, de confiance permettant d'analyser les messages échangés lors d'une exécution du protocole.

L'expérience a montré qu'il était nécessaire de disposer d'implémentations indépendantes et robustes pour étudier et comprendre les comportements d'un protocole donné, en particulier pour détecter et caractériser les anomalies. En effet, les implémentations disponibles de clients, serveurs ou dissecteurs sont parfois limitées (refus de certaines options), laxistes (acceptation silencieuse de paramètres erronés) ou fragiles (terminaison brutale du programme pour des valeurs inattendues, qu'elles soient licites ou non).

Ce constat a conduit au développement d'outils, l'objectif étant de développer *rapidement* des dissecteurs *robustes* et *efficaces*. Pour cela, plusieurs langages de programmation ont été successivement utilisés (C, C++, python, OCaml). L'objet de ce document est de présenter l'une de ces implémentations, reposant sur le pré-processeur `camlp4` d'OCaml, et de la comparer avec les autres. La section 1 présente l'historique du projet et deux applications concernant TLS et BGP. La section 2 donne un aperçu de l'ensemble des constructions apportées par Parsifal et la section 3 présente un exemple complet d'utilisation de Parsifal : le format d'archive TAR. La section 4 donne quelques chiffres pour comparer les programmes utilisant Parsifal avec d'autres implémentations équivalentes. Enfin, la section 4 conclut cette article et présente les perspectives d'évolution.

1. Historique du projet

1.1. Motivation initiale : analyser de nombreuses réponses TLS

Le point de départ de ces travaux était l'analyse d'une quantité importante (180 Go) de données récoltées concernant le protocole SSL/TLS. SSL (*Secure Sockets Layer*) et TLS (*Transport Layer Security*) sont deux variantes d'un même protocole, dont la dernière version est TLSv1.2 [1]. Leur objectif est de fournir un certain nombre de services pour sécuriser un canal de communication :

- authentification unilatérale ou mutuelle ;
- confidentialité des données échangées de bout en bout ;
- intégrité des données de bout en bout.

SSL a initalement été mis au point par Netscape pour protéger les connexions HTTP. Le résultat de cette association (HTTP + SSL) est désigné par l'acronyme HTTPS. Même si SSL/TLS peut être utilisé pour sécuriser d'autres protocoles applicatifs, HTTPS reste l'application principale de la couche cryptographique SSL/TLS. C'est la raison pour laquelle plusieurs équipes de recherche ont récemment entrepris des collectes sur l'ensemble des serveurs HTTPS accessibles sur internet, afin d'évaluer la qualité des réponses TLS apportées par tous ces serveurs.

Afin de pouvoir interpréter correctement les différents types de réponses, il est nécessaire de comprendre les différentes versions du protocole (SSLv2, SSLv3, TLSv1.0, TLSv1.1 et TLSv1.2), qui apportent chacune des subtilités dans les messages échangés. Parmi les messages pertinents pour ce type d'étude, le message `Certificate` contient la chaîne de certification permettant l'authentification du serveur. Elle est composée de certificats X.509 [2], dont le format repose sur l'ASN.1, une syntaxe pour encoder de manière structurée des entiers, des chaînes de caractères ou encore des objets binaires [1], ce qui apporte une complexité supplémentaire pour la dissection des messages.

Le laboratoire sécurité des réseaux et protocoles a mené des travaux sur de nombreuses traces HTTPS depuis deux ans. Les données utilisées proviennent de résultats mis à dispositions par l'Electronic Frontier Foundation [3, 4, 5] en 2010 et de collectes effectuées en 2011. Les traces analysées contiennent une très grande diversité de réponses, parfois incohérentes ou non conformes. Face à cet imposant corpus, il était difficile d'utiliser des outils existants pour extraire de manière fiable l'information pertinente pour les analyses. En effet, les implémentations existantes ont généralement un comportement inadapté à l'analyse, qu'il s'agisse d'un comportement limité (refus de certaines options), laxiste (acceptation silencieuse de paramètres erronés) ou fragile (terminaison brutale du programme pour des valeurs inattendues, qu'elles soient licites ou non).

Des outils spécifiques, maîtrisés par le laboratoire, ont donc été développés pour analyser ce grand volume de données.

1.2. Démarche de développement des outils

Le premier *parser* réalisé pour cette tâche a été écrit en Python. Nous avons ainsi obtenu rapidement un prototype pour extraire les premiers éléments des données. Cependant, ce premier programme s'est révélé trop lent face au volume à traiter.

Une seconde implémentation a donc vu le jour, en C++. Celle-ci reposait sur les *templates* et la programmation objet, et permettait d'obtenir un dissecteur flexible et efficace, mais au prix d'une grande quantité de code à écrire et d'erreurs de programmation parfois difficiles à diagnostiquer (fuites mémoire, erreurs de segmentation).

Pour pallier ce manque de robustesse, le développement d'une troisième version des outils a été entreprise, en OCaml. Afin de conserver la flexibilité imaginée pour le second prototype, un langage spécifique a été développé pour décrire les structures à disséquer. Les outils résultants étaient expressifs, efficaces et plus fiables que la version précédente. Malheureusement, à l'usage, si l'extensibilité était réelle, elle nécessitait des développements fastidieux.

Finalement, une nouvelle implémentation en OCaml a été réalisée pour réunir l'ensemble des qualités recherchées pour le développement de *parsers* :

- rapidité de développement ;
- garanties fortes sur la robustesse et la correction ;
- efficacité des programmes (en temps comme en consommation mémoire).

Cette quatrième mouture, baptisée Parsifal, résulte de la fusion entre le langage spécifique de description des objets analysés et le langage de programmation : on utilise directement le

1. Plus précisément, l'ASN.1 (*Abstract Syntax Notation One*) permet de décrire de manière abstraite la structure d'un certificat X.509 ; l'encodage utilisé pour les certificat est le DER (*Distinguished Encoding Rules*), une représentation concrète canonique de l'ASN.1.

langage OCaml pour les deux usages. Pour cela, le processeur `camlp4` est employé pour générer automatiquement des types et des fonctions à partir de descriptions brèves des structures à disséquer.

1.3. Autre application : MRT

BGP (*Border Gateway Protocol* [6]) est un autre protocole étudié par le laboratoire. Il s'agit du protocole de routage utilisé pour interconnecter les différents réseaux qui forment l'internet. Il existe de nombreuses RFC [2] décrivant BGP, ses extensions, ainsi que des protocoles connexes. En particulier, MRT (*Multi-Threaded Routing Toolkit* [7]) est un format d'échange utilisé pour archiver et transmettre l'ensemble des annonces vues par un routeur.

Certains projets proposent d'accéder aux informations de routage de l'internet collectées depuis plusieurs points dans le monde, et exportées au format MRT. Ces données sont d'un grand intérêt pour analyser l'état de l'internet, détecter et comprendre d'éventuels incidents affectant les informations de routage. L'observatoire de la résilience de l'internet français [8], mis en place par l'ANSSI en 2012, utilise de telles sources de données et doit disposer d'outils capables d'interpréter les fichiers MRT. Le projet RIS (*Routing Information Service*) fournit trois fois par jour une vision complète de 16 routeurs disséminés sur la planète, ce qui correspond à plusieurs centaines de méga-octets par jour, d'où le besoin d'avoir des dissecteurs efficaces.

Deux outils étaient utilisés au sein du laboratoire pour analyser cette grande quantité de données au format MRT. Le premier est un outil libre écrit en C, dont la qualité de code laisse à désirer [3]. Le second est un développement interne en OCaml, plus fiable que le premier.

Afin de valider la possibilité d'écrire rapidement avec Parsifal des *parsers* efficaces et robustes, un nouveau développement des outils d'analyse pour MRT a été réalisé. Il a suffi de 4 jours pour décrire en Parsifal les structures pertinentes pour les études menées et obtenir un outil indépendant des deux autres, plus rapide que l'implémentation en OCaml pré-existante et plus fiable que l'implémentation C.

2. Description des extensions du langage

L'idée générale de Parsifal est d'interpréter des descriptions courtes utilisant de nouveaux mots-clés (`enum`, `struct`, `union`, `alias`) pour générer automatiquement :

- des types OCaml décrivant les structures binaires correspondantes ;
- des fonctions pour disséquer les objets correspondants depuis une chaîne de caractère (`parse_t`) ou depuis un flux Lwt (`lwt_parse_t`) ;
- des fonctions pour exporter les objets sous forme binaire (`dump_t`) ou dans une représentation imprimable (`print_t`).

Le prototype de chacune de ces fonctions est :

```
val parse_t : string_input -> t
val lwt_parse_t : lwt_input -> t
val dump_t : t -> string
val print_t : ?indent:string -> ?name:string -> t -> string
```

Les fonctions `parse` travaillent sur des chaînes de caractères alors que `lwt_parse` utilisent les flux issus de la Lwt [9], une bibliothèque implémentant des threads synchrones à l'aide d'une interface monadique.

La fonction `print` accepte des options pour modifier l'indentation et le nom de la structure à afficher. De plus, les fonctions `parse`, `lwt_parse` et `dump` peuvent éventuellement accepter des paramètres supplémentaires.

2. Les RFC (*Request For Comments*) sont des documents publiés par l'IETF (*Internet Engineering Task Force*) qui décrivent entre autres les protocoles réseau de l'internet.

3. Cet outil en C se terminait parfois de manière brutale sur certains fichiers sans donner aucune explication.

Les sections suivantes décrivent les différentes constructions apportées par Parsifal, et des exemples permettant de comprendre ce qu'elles apportent. La grammaire détaillée est donnée en annexe.

2.1. Énumérations

La première construction ajoutée par Parsifal est l'énumération, qui ressemble à l'`enum` du C, le typage fort en plus. Par exemple, la déclaration suivante :

```
enum tls_version (16, UnknownVal V_Unknown, [with_lwt]) =
  | 0x0002 -> V_SSLv2, "SSLv2"
  | 0x0300 -> V_SSLv3, "SSLv3"
  | 0x0301 -> V_TLSv1, "TLSv1.0"
  | 0x0302 -> V_TLSv1_1, "TLSv1.1"
  | 0x0303 -> V_TLSv1_2, "TLSv1.2"
```

correspond au champ encodant la version du protocole SSL/TLS sur un entier 16 bits, et génère le type somme suivant :

```
type tls_version =
  | V_SSLv2 | V_SSLv3 | V_TLSv1 | V_TLSv1_1 | V_TLSv1_2
  | V_Unknown of int
```

ainsi que les fonctions suivantes :

```
val string_of_tls_version : tls_version -> string
val int_of_tls_version : tls_version -> int
val tls_version_of_int : int -> tls_version
val tls_version_of_string : string -> tls_version

val parse_tls_version : string_input -> tls_version
val lwt_parse_tls_version : lwt_input -> tls_version
val dump_tls_version : tls_version -> string
val print_tls_version : ?indent:string -> ?name:string -> tls_version -> string
```

En plus des quatre fonctions annoncées plus haut, les quatre premières fonctions générées permettent simplement de convertir le type énumération depuis et vers les entiers et les chaînes de caractères. Il est possible de décrire comment `tls_version_of_int` doit traiter une valeur non fournie dans l'énumération : dans l'exemple ci-dessus, `UnknownVal V_Unknown` indique qu'il faut renvoyer une valeur `V_Unknown i`, où `i` est la valeur non reconnue. L'autre comportement possible est de lever une exception en cas de motif inconnu.

2.2. PTypes

Avant de poursuivre avec les constructions apportées par Parsifal, il est nécessaire de décrire les PTypes, les types binaires compris par `struct`, `union` et `alias`.

Tout d'abord, Parsifal propose des types de base pour lesquels les implémentations des fonctions `parse`, `lwt_parse`, `dump` et `print` sont déjà disponibles : les entiers et les chaînes de caractères.

En plus de ces types scalaires, le PType `list` permet de décrire des listes d'objets du même type et `container` sert à encapsuler un sous-type en spécifiant la taille de la sous-chaîne à lire depuis l'entrée.

Pour les chaînes de caractères et les listes, les *parsers* générés par défaut essaient de lire l'ensemble de la chaîne qui leur est passée en entrée. On peut cependant préciser avec une expression entre parenthèses la taille en octets attendue pour une chaîne de caractères en octets ou le nombre d'éléments attendus pour une liste (`string(10)` correspond ainsi à une chaîne de 10 caractères).

Parfois, une chaîne de caractères ou une liste a une taille variable, spécifiée par un champ longueur le ou la précédent. Avec Parsifal, on peut spécifier à l'aide de crochets qu'un tel type est attendu :

`list [uint16] of uint16` permet de décrire une liste d'entiers sur 16 bits dont la longueur totale en octets est codées sur un entier 16 bits [4].

Enfin, n'importe quel type peut être utilisé comme PType personnalisé, dès lors qu'il s'agit d'un type OCaml, et que les fonctions `parse`, `dump` et `print` correspondantes existent. Il existe deux moyens de construire de nouveaux types :

- en écrivant un nouveau type OCaml et les fonctions nécessaires ;
- à l'aide de `struct` et `union`.

Pour illustrer le premier point, prenons un exemple tiré de MRT, présenté dans la section 1.3. De nombreux champs du protocole BGP sont des adresses IPv4 et IPv6 encodées directement en binaires, respectivement sur 32 et 128 bits. Le code suivant montre comment ajouter `ipv4` à l'arsenal des PTypes disponibles :

```
type ipv4 = string

let parse_ipv4 input = parse_string 4 input
let lwt_parse_ipv4 input = lwt_parse_string 4 input

let dump_ipv4 ipv4 = ipv4

let print_ipv4 ?indent:(indent="") ?name:(name="ipv4") s =
  let res = string_of_ipv4 s in
  Printf.sprintf "%s%s: %s\n" indent name res
```

L'autre manière de créer un PType consiste à utiliser les mots-clés `struct` et `union` pour décrire des types complexes à partir de PTypes existant, qui sont décrits dans les paragraphes suivants.

2.3. Structures

Les structures que l'on peut décrire avec Parsifal sont des enregistrements qui viennent avec des fonctions `parse`, `dump` et `print` auto-générées. Afin de présenter le fonctionnement du mot-clé `struct`, on peut s'intéresser à la manière d'implémenter les messages d'alerte TLS. Ces messages sont spécifiés de la manière suivante dans TLSv1.2 [1] (pages 27 et 28) :

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
     ...
    unsupported_extension(110),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

Ces messages sont simplement composés d'un octet décrivant le niveau de l'alerte et d'un second octet décrivant le type d'alerte. Pour implémenter cette spécification en Parsifal, il suffit de créer deux types énumérations sur 8 bits, puis de créer un type enregistrement, comme dans le code suivant :

4. Ce type particulier peut servir à décrire la liste des suites cryptographiques (*ciphersuites*) proposées par un client dans le protocole TLS.

```
enum tls_alert_level (8, UnknownVal AL_Unknown, []) =
  | 1 -> AL_Warning, "Warning"
  | 2 -> AL_Fatal, "Fatal"

enum tls_alert_type (8, UnknownVal AT_Unknown, []) =
  | 0 -> AT_CloseNotify, "CloseNotify"
  | 10 -> AT_UnexpectedMessage, "UnexpectedMessage"
    ...
  | 110 -> AT_UnsupportedExtension, "UnsupportedExtension"

struct tls_alert =
{
  alert_level : tls_alert_level;
  alert_type : tls_alert_type
}
```

Une fois ces quelques lignes écrites, qui sont essentiellement une reformulation de la spécification, il est directement possible d'utiliser les fonctions générées telles que `parse_tls_alert`.

2.4. Unions

Empruntant encore à la terminologie du langage C, les unions de Parsifal sont en réalité des types somme augmentés par un constructeur par défaut. L'idée est d'utiliser un discriminant passé en argument à la fonction `parse` pour en déduire le constructeur du type somme à utiliser. Ce discriminant va être comparé aux différents motifs proposés dans la description de l'union, et le constructeur adapté sera choisi.

Pour illustrer ce concept, prenons un exemple issu du format MRT. Pour décrire les différents systèmes interconnectés par BGP, on utilise la notion de système autonome (AS, *Autonomous Systems*), un numéro identifiant un opérateur. Initialement, ces AS étaient représentés sur 16 bits, mais cela n'est plus suffisant désormais et les AS sont parfois décrits sur 32 bits. L'exemple suivant montre comment décrire un AS avec une union, et comment employer le type généré dans une structure :

```
union autonomous_system (UnparsedAS, [enrich]) =
  | 16 -> AS16 of uint16
  | 32 -> AS32 of uint32

struct bgp_as_path_segment [param as_size] =
{
  path_segment_type : uint8;
  path_segment_length : uint8;
  path_segment_value : list(path_segment_length) of autonomous_system(as_size)
}
```

Le type `autonomous_system` sera vu comme un entier 16 ou 32 bits selon le discriminant qui lui est passé. La déclaration de la structure `bgp_as_path_segment` conduit à la génération d'une fonction `parse_bgp_as_path_segment` qui prend un argument supplémentaire (`as_size`), puis tente de lire le type de segment (un octet), la longueur du segment (un autre octet), puis une liste contenant `path_segment_length` éléments du type union défini. On remarquera que la valeur `as_size` est alors passée comme discriminant à `parse_autonomous_system`.

Voici des extraits du code OCaml généré par Parsifal à partir des deux types ci-dessus :

```
type autonomous_system = | AS16 of int | AS32 of int | UnparsedAS of string

let parse_autonomous_system discriminator input =
  match discriminator with
    | 16 -> AS16 (parse_uint16 input)
    | 32 -> AS32 (parse_uint32 input)
```

```
    | _ -> UnparsedAS (parse_rem_string input)

type bgp_as_path_segment =
{
  path_segment_type : int;
  path_segment_length : int;
  path_segment_value : autonomous_system list
}

let parse_bgp_as_path_segment as_size input =
  let path_segment_type = parse_uint8 input in
  let path_segment_length = parse_uint8 input in
  let path_segment_value = parse_list path_segment_length
      (parse_autonomous_system as_size) input
  in
    {
      path_segment_type = path_segment_type;
      path_segment_length = path_segment_length;
      path_segment_value = path_segment_value;
    }
```

2.5. Autres constructions

Parsifal propose d'autres mots-clés et fonctionnalités, qui ne seront pas décrits en détails ici :
- pour permettre de produire simplement des fonctions `parse`, `dump` et `print` pour des listes, on peut déclarer des alias avec le mot clé `alias` ;
- afin de permettre la description de structures ASN.1 encodées au format DER (par exemple les certificats X.509), Parsifal fournit des types prédéfinis et un mot-clé `asn1_alias` ;
- parfois, la dissection de messages complexes ne requiert pas une dissection en profondeur de toutes les structures. Afin d'accélérer les outils, il est possible de désactiver l'enrichissement des unions non pertinentes à l'aide d'un paramètre `enrich`.

3. Exemple d'un développement avec Parsifal : le format TAR

3.1. Présentation du format

TAR est un format d'archive classique utilisé dans le monde Unix. Son rôle est d'agréger un ensemble de fichiers à archiver dans un seul gros fichier, qui pourra éventuellement être compressé ensuite.

Une archive s'organise en une succession d'entrées, dont le contenu est décrit dans la table 1. Chaque entrée commence par un en-tête dont les champs varient en fonction de la version du format utilisée ; la table 2 décrit deux de ces versions : le format original (colonne TAR) et une variante plus récente (colonne `ustar`). L'en-tête est contenu dans un « bloc » de 512 octets, et le contenu du fichier qui suit l'en-tête est lui-aussi complété par des caractères nuls pour former des blocs de 512 octets.

TAR présente une particularité quant au stockage des valeurs entières : celles-ci sont systématiquement stockées sous la forme d'une chaîne de caractère représentant la valeur numérique en octal. Cette méthode permet d'éviter des difficultés d'encodage telles que la prise en compte de l'*endianness*, mais au prix d'une plus grande consommation mémoire.

Le champ à l'offset 156 décrit le type de fichier. La table 3 récapitule les différentes valeurs qu'il peut prendre (dont certaines sont uniquement accessibles avec le format étendu `ustar`). En fonction de la valeur prise par ce champ, certains des champs de l'en-tête ne sont pas pertinents et devront être remplis de caractères nuls.

Offset	Long.	Description
0	512	En-tête TAR, complété par des zéros
512	*« Taille du fichier »* alignée à 512 octets	Contenu du fichier, complété par des zéros

TABLE 1 – Description d'une entrée TAR.

Offset	Long.	Description	
		TAR	`ustar`
0	100	Nom du fichier	
100	8	Permissions	
108	8	UID	
116	8	GID	
124	12	Taille du fichier	
136	12	*Timestamp* de la dernière modification	
148	8	Somme de contrôle de l'en-tête	
156	1	Indicateur de lien	Type de fichier
157	100	Nom du fichier pointé par le lien	
257	6	-	Indicateur `"ustar"`
263	2	-	Version `ustar` ("00")
265	32	-	Propriétaire
297	32	-	Groupe propriétaire
329	8	-	Numéro majeur du périphérique
337	8	-	Numéro mineur du périphérique
345	155	-	Préfixe

TABLE 2 – Description de l'en-tête TAR.

Caractère	Description	Spécifique à `ustar`
`<NUL>`, `0`	fichier ordinaire	-
`1`	lien dur	-
`2`	lien symbolique	-
`3`	périphérique en mode caractères	oui
`4`	périphérique en mode blocs	oui
`5`	répertoire	oui
`6`	file FIFO	oui
`7`	fichier contigu	oui

TABLE 3 – Valeurs du champ « Indicateur de lien/Type de fichier ».

3.2. Une première implémentation

Afin d'utiliser Parsifal pour décrire les fichiers au format TAR, commençons par décrire les valeurs du champ « Indicateur de lien/Type de fichier ». Il s'agit d'une énumération sur 8 bits qui s'écrit logiquement avec le nouveau mot-clé `enum` :

```
enum file_type (8, UnknownVal UnknownFileType, []) =
  | 0 -> NormalFile
  | 0x30 -> NormalFile
  | 0x31 -> HardLink
  | 0x32 -> SymbolicLink
  | 0x33 -> CharacterSpecial
  | 0x34 -> BlockSpecial
  | 0x35 -> Directory
  | 0x36 -> FIFO
  | 0x37 -> ContiguousFile
```

Ensuite, on peut décrire l'enregistrement contenant les différents champs constituant d'un en-tête à l'aide du mot clé `struct` de la manière suivante :

```
struct tar_header =
{
  file_name : string(100);
  file_mode : string(8);
  owner_uid : string(8);
  owner_gid : string(8);
  file_size : string(12);
  timestamp : string(12);
  checksum  : string(8);
  file_type : file_type;
  linked_file : string(100);
  ustar_magic : magic("ustar\x0000");
  owner_user : string(32);
  owner_group : string(32);
  device_major : string(8);
  device_minor : string(8);
  filename_prefix : string(155);
  hdr_padding : binstring(12);
}
```

On obtient alors un type enregistrement `tar_header`. Il contient en particulier un champ `file_size` de type chaîne de caractères, encodant la taille du fichier en octal. Comme la valeur entière est nécessaire pour savoir combien d'octets lire pour le contenu du fichier, il nous faut une fonction de conversion[5] :

```
let int_of_tarstring octal_value =
  let len = String.length octal_value in
  if len = 0
  then 0
  else begin
    let real_octal_value = String.sub octal_value 0 (len -1) in
    int_of_string ("0o" ^ octal_value)
  end
```

Il ne nous reste plus qu'à finaliser l'implémentatin en décrivant ce qu'est une entrée TAR et en créant un alias décrivant un fichier :

```
struct tar_entry =
{
  header : tar_header;
  file_content : binstring(int_of_tarstring header.file_size);
```

5. Pour des raisons historiques, les chaînes de caractères décrivant la représentation en octal des valeurs numériques se terminent par un espace ou un caractère nul, ne laissant ainsi en pratique que 7 octets pour encoder la valeur.

```
  file_padding : binstring(512 - ((int_of_tarstring header.file_size) % 512))
}

alias tar_file [with_lwt] = list of tar_entry;
```

Une fois ces quelques dizaines de lignes écrites, on dispose d'une implémentation fonctionnelle pour disséquer, produire et afficher des fichiers TAR. Il est par exemple possible d'utiliser les fonctions générées par le pré-processeur pour lister le nom des fichiers contenus dans une archive TAR. On utilise pour cela la fonction `lwt_parse_tar_file` :

```
let handle_filename filename =
  Lwt_unix.openfile filename [Unix.O_RDONLY] 0 >>= fun fd
  lwt_parse_tar_file (input_of_fd filename fd) >>= fun tar_file
  let print_filename entry = print_string entry.file_name in
  List.iter print_filename tar_file;
  return ()

let _ =
  Lwt_unix.run (handle_filename "test.tar");
```

Une fois ce programme compilé avec le pré-processeur `camlp4` et lié avec les bibliothèques décrivant les fonctions de base, on peut lancer le programme pour décrire le contenu de l'archive `test.tar`.

3.3. Traitement des vieilles archives

Le type pré-défini `magic` utilisé dans la structure `tar_header` permet de lire un marqueur attendu (parfois appelé *magic* en anglais). Si le marqueur ne peut être trouvé, la dissection échoue sur une erreur.

La première version proposée est fonctionnelle, mais si l'on souhaite proprement prendre en compte les vieilles archives, ne contenant pas l'en-tête étendu commençant par le marqueur `ustar`, on doit réécrire la description de l'en-tête de la manière suivante, en séparant l'en-tête en deux morceaux, dont un est déclaré optionnel. Pour cela, on remplace les déclarations de `tar_header` et `tar_entry` par le code suivant :

```
struct ustar_header =
{
  ustar_magic : magic("ustar\x0000");
  owner_user : string(32);
  owner_group : string(32);
  device_major : string(8);
  device_minor : string(8);
  filename_prefix : string(155);
}

struct tar_header =
{
  file_name : string(100);
  file_mode : string(8);
  owner_uid : string(8);
  owner_gid : string(8);
  file_size : string(12);
  timestamp : string(12);
  checksum  : string(8);
  file_type : file_type;
  linked_file : string(100);
  optional ustar_header : ustar_header;
  hdr_padding : binstring;
}
```

```
struct tar_entry =
{
  header : container(512) of tar_header;
  file_content : binstring(int_of_tarstring header.file_size);
  file_padding : binstring(512 - ((int_of_tarstring header.file_size) % 512))
}
```

Avec cette seconde version, on utilise un conteneur de 512 octets pour lire l'en-tête. Si l'en-tête est étendu, le champ `ustar_header` sera complété par la fonction `parse_ustar_header` et le champ `hdr_padding`, une chaîne binaire consommant les octets restants du conteneur, contiendra 12 octets ; dans le cas contraire, la fonction `parse_ustar_header` échouera renvoyant `None` pour le champ `ustar_header` et `hdr_padding` aura une longueur de 255 octets.

3.4. Prise en compte transparente des valeurs numériques octales

Le format TAR faisant un usage important des chaînes de caractères encodant une valeur numérique en octal. Les deux implémentations proposées ci-dessus encodent simplement ces champs comme une chaîne de caractères d'une taille donnée. Afin d'améliorer l'ergonomie des types construits, on pourrait créer de toute pièce un nouveau type, `tar_numstring` pour prendre en compte ce format particulier. Il suffit pour cela de définir un nouveau type OCaml et les fonctions associées :

```
type tar_numstring = int

let parse_tar_numstring len input =
  let octal_value = parse_string (len - 1) input in
  drop_bytes 1 input;
  int_of_string ("0o" ^ octal_value)

let dump_tar_numstring len v =
  Printf.printf "%*.*o\x00" len len v

let print_tar_numstring ?indent:(indent="") ?name:(name="numstring") v =
  Printf.sprintf "%s%s: %d (%o)\n" indent name v v
```

Une fois ce nouveau PType défini, on peut modifier les champs représentant des valeurs numériques en octal (comme `file_size`) en replaçant le type `string(n)` par `tar_numstring(n)`. Cela nécessite de reprendre la définition de `ustar_header` et de `tar_header` ; ensuite, la description de `tar_entry` se simplifie puisque `header.file_size` est désormais de type `tar_numstring`, c'est-à-dire un entier [6] :

```
struct tar_entry =
{
  header : container(512) of tar_header;
  file_content : binstring(header.file_size);
  file_padding : binstring(512 - (header.file_size % 512))
}
```

4. Quelques résultats

Parsifal fournit les PTypes suivants : les types scalaires de base (entiers, chaînes de caractères), les types liste et conteneur, quelques types utiles (`magic`, `ipv4`, etc.) et les types ASN.1 de base. Pour cela, l'implémentation repose sur

- des pré-processeurs `camlp4`, qui comportent environ 1200 lignes de code ;
- des bibliothèques annexes pour implémenter la gestion des PTypes, sur 1300 lignes de code.

Comme indiqué plus haut, plusieurs formats et protocoles suivants ont été implémentés à partir de ces briques de base :

6. La fonction `int_of_tarstring` définie plus haut devient alors inutile.

– les certificats X.509 (100 lignes) ;
– les messages du protocole TLS (600 lignes) ;
– les messages du protocole MRT (300 lignes) ;
– le format TAR, pour les besoins de cet article (moins de 100 lignes).

Une fois ces formats et protocoles décrits, il est possible d'écrire des outils utilisant les fonctions `parse`, `dump` et `print`. Parmi ces outils, citons-en deux issus des études décrites dans les sections 1.1 et 1.3 :

– un outil pour extraire les certificats des réponses TLS (`answer2certs`) ;
– un programme pour afficher les annonces de route BGP à partir d'un fichier MRT (`mrtdump`).

Pour ces deux applications, nous disposons de plusieurs implémentations dans différents langages. Le tableau suivant récapitule le nombre de lignes et le temps d'exécution sur des données caractéristiques des différentes implémentations :

	C ou C++	OCaml	OCaml + Parsifal[7]
`answer2certs`	Dev. interne (C++)	Dev. interne	Dev. interne
Nb lignes	8 500	4 000	1 000
Temps	100 s	40 s	8 s
`mrtdump`	`libbgpdump`[8]	Dev. interne	Dev. interne
Nb lignes	4 000	1 200	550
Temps	23 s	180 s	35 s

Dans tous les cas, on constate que l'implémentation utilisant Parsifal requiert bien moins de lignes de code que les implémentations équivalentes étudiées. Du point de vue des performances, il est intéressant de constater que l'implantation initiale en C++ de `answer2certs` n'était pas très bonne, alors que sa réécriture en OCaml puis avec Parsifal se sont révélées non seulement plus succinctes, mais aussi beaucoup plus rapide ; en revanche, l'implémantation disponible sur étagère pour disséquer des fichiers MRT en C est 33 % plus rapide que la version reposant sur Parsifal, mais les deux versions ne sont pas tout à fait iso-fonctionnelles, puisque certains fichiers MRT font échouer `libbgpdump` alors que nos développements les traitent correctement.

5. Conclusion et perspectives

Pour remplir sa mission de compréhension des protocoles réseau, le laboratoire sécurité des réseaux et protocoles a développé plusieurs outils de dissection de protocoles. Parmi ceux-ci, Parsifal est une implémentation générique de *parsers* reposant sur un pré-processeur `camlp4` et sur une bibliothèque auxiliaire. Avec ses 2500 lignes de code, Parsifal permet de générer des dissecteurs réunissant toutes les propriétés recherchées (rapidité de développement, garanties fortes sur la robustesse et la correction et efficacité des programmes).

Deux projets écrits en Python présentent des similarités avec les travaux exposés ici :

– `scapy` [10], une boîte à outils pour manipuler les paquets réseau ; ce projet permet également de définir rapidement de nouveaux protocoles réseau pour étendre la boîte à outils ;
– `hachoir` [11], une bibliothèque pour écrire rapidement des dissecteurs pour des formats de fichiers.

Le langage Python semble un candidat naturel pour réaliser ce genre de projet, grâce à ses capacités d'introspection. Il est intéressant de remarquer que l'utilisation d'un pré-processeur OCaml permet d'arriver au même résultat.

7. Pour la colonne « OCaml + Parsifal », le décompte ne comprend ni le pré-processeur ni les fonctions auxiliaires de base.
8. Le projet `bgpdump` est disponible à l'adresse `www.ris.ripe.net/source/bgpdump`.

Parmi les développements prévus pour Parsifal, les pistes suivantes sont envisagées :
- détailler les extensions des certificats X.509 qui ne sont que partiellement enrichies aujourd'hui ;
- écrire et animer l'automate du protocole de négociation TLS (à ce jour, seule une animation rudimentaire a été déveoppée) ;
- pour disposer d'une pile TLS complète, ajouter les appels aux fonctions cryptographiques pour mettre réellement en œuvre le tunnel négocié ;
- décrire de nouveaux formats de fichiers et de nouveaux protocoles (DNS, OCSP) ou formats de fichiers (fichiers de traces réseau `pcap`, format exécutable PE) ;

Références

[1] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[2] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[3] Electronic Frontier Foundation. The EFF SSL Observatory. `https://www.eff.org/observatory`, 2010-2012.

[4] P. Eckersley and J. Burns. An Observatory for the SSLiverse, Talk at Defcon 18, 2010.

[5] P. Eckersley and J. Burns. Is the SSLiverse a safe place ?, Talk at 27C3, 2010.

[6] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFC 6286.

[7] L. Blunk, M. Karir, and C. Labovitz. Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format. RFC 6396 (Proposed Standard), October 2011.

[8] ANSSI and AFNIC. Rapport 2011 de l'observatoire de la résilience de l'internet français. `www.ssi.gouv.fr/IMG/pdf/rapport-obs-20120620.pdf`, 2012.

[9] J. Vouillon. Lwt : a Cooperative Thread Library. `http://ocsigen.org/lwt/`, 2002-2012.

[10] P. Biondi and the Scapy community. Scapy. `http://www.secdev.org/projects/scapy/`, 2003-2012.

[11] V. Stinner. Hachoir. `https://bitbucket.org/haypo/hachoir/wiki/Home`, 2009-2012.

A. Annexe : grammaire des extensions

A.1. Options

option ::= `with_lwt`
| `with_exact` [9]
| `top` [9]
| `param` *expression* [9]
| `enrich` [10]
| `exhaustive` [10]

9. Ces options n'ont de sens que pour les structures, les unions et les alias.
10. Ces options n'ont de sens que pour les unions.

A.2. PTypes

PType ::= `char`
| `uintN`
| `string`
| `string` (*expression*)
| `string` [*type entier*]
| `binstring`
| `binstring` (*expression*)
| `binstring` [*type entier*]
| `list of` *PType*
| `list` (*expression*) `of` *PType*
| `list` [*type entier*] `of` *PType*
| `container` (*expression*) `of` *PType*
| `container` [*type entier*] `of` *PType*
| *type personnalisé*
| *type personnalisé* (*paramètres*)

A.3. Énumérations

énumération ::= `enum` *identifiant* (taille : *int*, *comportement_enum*, *option list*) =
| *motif* → *Constructeur* [, *texte*] [11]
...
| *motif* → *Constructeur* [, *texte*]

comportement_enum ::= `UnknownVal` *Constructor*
| `Exception` *Exception*

A.4. Structures, unions et alias

structure ::= `struct` *identifiant* [*option list*] = {
[`optional`] *identifiant* : *PType* ;
...
[`optional`] *identifiant* : *PType*
}

union ::= `union` *identifiant* (*comportement_union*, [*option list*]) =
| *motif* → *Constructeur* [`of` *PType*]
...
| *motif* → *Constructeur* [`of` *PType*]

comportement_union ::= *Constructor*
| *Constructor* `of` *PType*

alias ::= `alias` *identifiant* = *PType*

11. Ce texte sera renvoyé par `string_of_enum` pour cette valeur de l'énumération. Si aucun texte n'est donné, le nom du constructeur sera utilisé automatiquement à la place.

A.5. Description des objets ASN.1

ASN1Header ::= *ASN1Class* `*` *ASN1Tag*

ASN1Type ::= *ASN1Header* `option` `*` *ASN1TypeContent*

ASN1TypeContent ::= `asn1_bool`
| `asn1_integer`
| `asn1_bitstring`
| `asn1_enumerated`
| `asn1_octetstring`
| `asn1_null`
| `asn1_oid`
| `asn1_list of` *ASN1Type*
| `asn1_container of` *ASN1Type*
| `asn1_anything`
| *custom type*
| *custom type* (*paramètres*)

alias_ASN.1 ::= `asn1_alias` *identifiant* = *ASN1Type*