Réflexions sur l'écriture de *parsers* binaires robustes et présentation de la solution Parsifal

Olivier Levillain[1,2]

1: Agence nationale de la sécurité des systèmes d'information (ANSSI)
2: Télécom Sud Paris

Dans le cadre de ses activités d'expertise, le laboratoire sécurité des réseaux et protocoles de l'ANSSI est amené à étudier divers protocoles de communication. L'étude fine de ces protocoles passe par l'utilisation de *parsers* (ou dissecteurs) permettant d'analyser les messages binaires échangés lors d'une exécution du protocole. L'expérience a montré qu'il fallait disposer d'outils robustes et maîtrisés pour étudier et comprendre les comportements d'un protocole donné, en particulier pour en détecter et caractériser les anomalies. En effet, les implémentations disponibles 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 nous a conduit au développement d'outils, l'objectif étant de développer *rapidement* des dissecteurs *robustes* et *performants*. Pour cela, plusieurs langages de programmation ont été successivement utilisés (C, C++, Python, OCaml). L'objet de ce document est de présenter les différents choix explorés, en insistant sur Parsifal, une implémentation reposant sur le pré-processeur `camlp4` d'OCaml, et de décrire le retour d'expérience de l'utilisation de Parsifal pour l'écriture d'outils pertinents pour la sécurité des systèmes d'information.

1. Introduction

Afin de mieux comprendre un protocole et la manière dont il est utilisé *in vivo*, le laboratoire sécurité des réseaux et protocoles de l'ANSSI s'intéresse notamment à l'analyse de grands volumes de données issus de mesures réalisées sur internet :

— en 2010, l'EFF (*Electronic Frontier Foundation*) a mis à disposition le résultat d'un ensemble important de traces réseau représentant des échanges suivant le protocole TLS [10, 11]. Des mesures complémentaires ont été effectuées par l'ANSSI et Télécom Sud Paris, et ont fait l'objet d'une publication [13]. Les données correspondantes représentent environ 180 Go ;

— pour mieux comprendre la structure de l'internet français, l'ANSSI a créé en 2012 l'Observatoire de la résilience de l'internet français [4], en partenariat avec l'Afnic [1]. Plusieurs indicateurs définis dans les rapports de l'Observatoire traitent de données BGP, recueillies par des collecteurs du RIS [2]. Ces données représentent environ 2 Go de données par jour.

L'analyse de ces données pose plusieurs difficultés. Tout d'abord, les fichiers à analyser représentent un volume conséquent. Ensuite, les informations à extraire sont contenues dans des messages de protocoles complexes (voir section 2.1 pour un bref aperçu du protocole TLS). Enfin, il s'agit de données brutes, non filtrées, dont la qualité, voire l'innocuité, laisse parfois à désirer.

Depuis 2009, l'ANSSI a coordonné deux études sur la question de l'adéquation de différents langages de programmation au développement d'applications de sécurité [2, 3, 17, 1]. Dans le cas particulier

1. L'Afnic est l'association française pour le nommage Internet en coopération.
2. Le *Routing Information Service* est un projet visant à collecter les informations de routage de l'internet.

des *parsers* de protocoles binaires, les principales propriétés attendues sont : la possibilité de traiter des *structure complexes*, la *rapidité d'écriture*, les *performances* et la *robustesse*. Ainsi, afin d'extraire de l'information pertinente de cet imposant corpus, plusieurs implémentations ont été réalisées, en utilisant des langages différents.

L'étude de nombreux formats binaires (protocoles réseau, formats d'image, formats d'exécutables) nous a permis de mieux comprendre les besoins logiciels des *parsers* binaires : les mille et une manières de représenter un entier [3], champs de taille variable, champs dépendants d'un discriminant, dépendances vis-à-vis d'un contexte plus ou moins complexe, *parsing* non linéaire (c'est-à-dire reposant sur des pointeurs au sein du message). Dans cet article, notre contribution est double : premièrement, à partir de ces connaissances, nous présentons différentes manières de *parser* les formats binaires pour les comparer en fonction de nos besoins ; deuxièmement, nous décrivons Parsifal, un outil développé à l'ANSSI pour aider à l'écriture de *parsers* binaires.

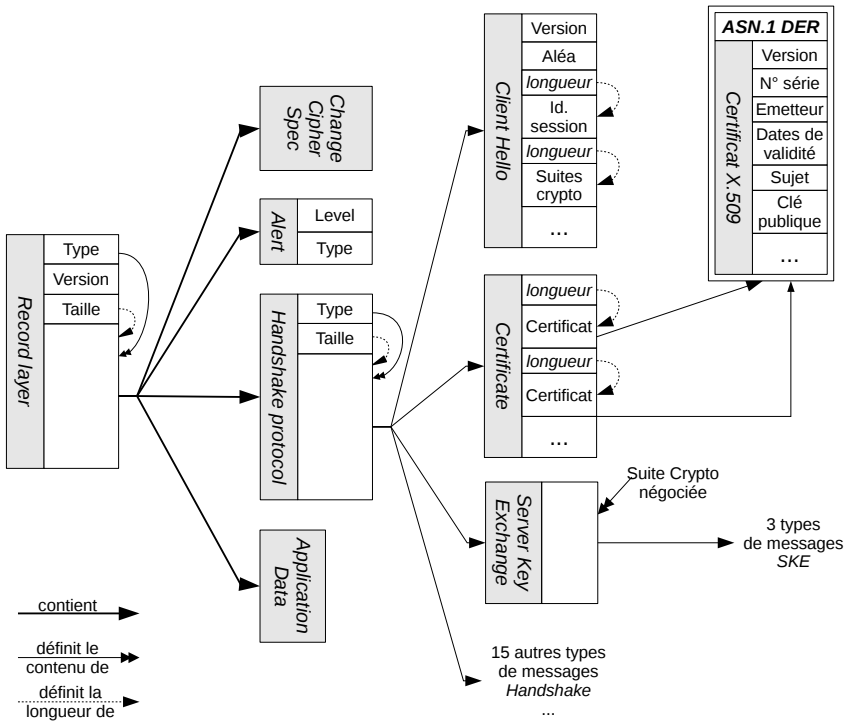

FIGURE 1 – Description d'une partie des messages TLS.

2. Différentes manières de *parser* des formats binaires

Dans un premier temps, le format des messages TLS est brièvement décrit, pour illustrer les besoins rencontrés dans l'écriture de *parsers* binaires. Dans un second temps, diverses solutions pour écrire des *parsers* sont présentées et étudiées. Pour chaque méthode décrite, un exemple de code permettant de *parser* une alerte TLS sera donné.

3. Au-delà des classiques entiers *big-* ou *little-endian*, l'encodage DER de l'ASN.1 utilise trois autres représentations, et les archives TAR utilisent une chaîne de caractères représentant la valeur octale !

2.1. Exemple d'un format binaire : le protocole TLS

Le point de départ de ces travaux était l'analyse d'une quantité importante de données récoltées représentant des messages du protocole SSL/TLS, deux variantes d'un même protocole, dont la dernière version est TLS 1.2 [9]. Ce protocole a pour objectif d'établir un canal de communication sécurisé entre un client et un serveur et représente aujourd'hui la solution la plus employée sur internet pour protéger des échanges.

Afin de pouvoir interpréter correctement les différents types de réponses, il est nécessaire de comprendre les différentes versions du protocole, qui apportent chacune leur lot de subtilités dans les messages échangés. La figure 1 donne un aperçu des structures décrivant les messages TLS. En particulier, certains champs ont une taille variable, d'autres dépendent d'un champ précédent (un discriminant), voire du contexte provenant des messages précédents. De plus, TLS utilise des certificats X.509, qui reposent sur l'encodage DER, différent de l'encodage du reste des messages [4].

2.2. Méthode 1 : conversion directe du flux binaire en utilisant des structures bas-niveau (*cast*)

Description Une manière simple d'interpréter un format binaire est de plaquer sur le contenu binaire à analyser la structure des données attendues. C'est une approche classique en C, qui pour un *record* TLS revient à utiliser le code suivant :

```
typedef struct __attribute__((packed)) {
  uint8_t alert_level;
  uint8_t alert_type;
} tls_alert;

typedef union __attribute__((packed)) {
  tls_alert alert;
  // Autres types de messages...
} record_content;

typedef struct __attribute__((packed)) {
  uint8_t        content_type;
  uint16_t       tls_version;
  uint16_t       record_length;
  record_content record_content;
} tls_record;

tls_record r;
read (f, &r, 5);
read (f, &r.record_content, r.record_length);
```

Analyse Les avantages de cette solution sont la simplicité d'écriture et son efficacité. Elle s'avère cependant inexploitable face à des formats complexes comme TLS, puisqu'à chaque champ de taille variable et à chaque type somme reposant sur un discriminant, il faut analyser le champ en question et plaquer une nouvelle structure en conséquence.

De plus, on n'a ici aucune garantie que les champs remplis aient une valeur valide (par exemple, seules certaines valeurs sont acceptables pour le champ `content_type`). Enfin, en termes de robustesse, de nombreuses erreurs de programmation sont possibles, pouvant mener par exemple à des débordements de tampon. Cette méthode n'étant clairement pas adaptée à l'analyse de protocoles complexes comme TLS, aucune implémentation utilisant cette technique n'a vraiment été testée.

4. Ainsi, le message TLS contenant les certificats peut contenir jusqu'à 15 niveaux de structures imbriquées.

2.3. Méthode 2 : interprétation d'un langage de description (DSL)

Description Il est également possible de créer un nouveau langage, un DSL[5], pour décrire les structures à *parser*, puis d'écrire un interpréteur pour ce DSL. Cette méthode est exploitée par Scapy, une boîte à outil pour analyser et forger des paquets réseau. En utilisant la syntaxe de cet outil, le code précédent s'écrit de la manière suivante :

```
_tls_type = { 20: "Change_Cipher_Spec", 21: "Alert",
              22: "Handshake",          23: "Application_data" }

_tls_version = { 0x0002: "SSLv2",   0x0300: "SSLv3",
                 0x0301: "TLSv1",   0x0302: "TLSv1.1",
                 0x0303: "TLSv1.2" }

_tls_alert_level = { ... }   # Deux autres
_tls_alert_type = { ... }    # énumérations

class TLSPlaintext():
    name = "TLS_Plaintext"
    fields_desc = [ ByteEnumField("type", None, _tls_type),
                    ShortEnumField("version", None, _tls_version),
                    FieldLenField("len", None, length_of="fragment", fmt="!H"),
                    StrLenField("fragment", "", length_from=lambda pkt: pkt.length) ]

class TLSAlert():
    name = "TLS_Alert"
    firlds_desc = [ ByteEnumField("alert_level", None, _tls_alert_level),
                    ByteEnumField("alert_type", None, _tls_alert_type) ]

bind_layers (TLSPlaintext, TLSAlert, type=21)
```

Analyse Cette solution élégante et concise permet aisément de décrire des formats mettant en œuvre des champs de longueur variable ou des types dépendant d'un champ précédent. Cependant, l'étage d'interprétation peut se révéler coûteux en performance à l'exécution (surtout dans un langage interprété tel que Python).

Deux outils écrits en Python utilisent cette méthode :

— Scapy [6], une boîte à outils pour manipuler les paquets réseau, qui permet également de définir de nouveaux protocoles réseau pour étendre la boîte à outils ;

— Hachoir [18], une bibliothèque pour écrire des dissecteurs pour des formats de fichiers binaires.

Dans le cadre de nos travaux, des solutions utilisant Python ont été envisagées. En effet, ce langage autorise l'introspection, qui permet de mélanger plus facilement le DSL au langage de programmation. Cependant, les programmes résultants se sont révélés trop lents à l'exécution. De plus, le langage Python ne permet pas d'obtenir de garantie forte sur le typage des données, ce qui n'était pas en accord avec notre objectif de robustesse.

C'est pourquoi un prototype reposant sur un DSL a été écrit en OCaml, mais plusieurs difficultés ont été rencontrées, quant à la définition du langage de manipulation des valeurs *parsées*. En effet, le DSL doit-il se contenter de décrire les structures à *parser* ? Ou doit-il définir également les traitements à réaliser ? Cette approche a finalement été abandonnée au profit de la méthode 5, définie plus loin.

2.4. Méthode 3 : écriture manuelle des types et des fonctions (Manuelle)

Description Si l'on souhaite conserver une grande expressivité dans les formats analysables par nos outils, l'écriture manuelle reste toujours possible. Pour cela, on écrit des fonctions de bibliothèque pour réduire les tâches répétitives (lire un entier, lire une chaîne de longueur donnée, lire et valider

5. *Domain Specific Language.*

une valeur énumérée, etc.). Le code correspondant est cette fois beaucoup plus long, puisqu'il faut décrire entièrement les structures et les méthodes `parse` :

```
type tls_content_type =
  | CT_ChangeCipherSpec     | CT_Alert
  | CT_Handshake            | CT_ApplicationData

type tls_version = SSLv2 | SSLv3 | TLSv1 | TLSv1_1 | TLSv1_2

type alert_level = ...
type alert_type = ...

type tls_alert = {
  alert_level : alert_level;
  alert_type : alert_type;
}

type record_content = Alert of tls_alert | ...

type tls_record = {
  content_type : tls_content_type;
  record_version : tls_version;
  record_content : record_content;
}

let parse_tls_version input =
  match (parse_uint16 input) with
  | 0x0002 -> SSLv2
  | 0x0300 -> SSLv3
  | 0x0301 -> TLSv1
  | 0x0302 -> TLSv1_1
  | 0x0303 -> TLSv1_2
  | _ -> raise ...

let parse_tls_content_type input = ... (* 3 autres fonctions *)
let parse_alert_level input = ...      (* pour parser des    *)
let paree_alert_type input = ...       (* énumérations       *)

let parse_tls_alert input =
  let l = parse_alert_level input in
  let t = parse_alert_type input in
  { alert_level = l; alert_type = t }

let parse_record_content content_type input =
  match content_type with
  | CT_Alert -> Alert (parse_tls_alert input)
  | ...

let parse_tls_record context input =
  let content_type = parse_tls_content_type input in
  let record_version = parse_tls_version input in
  let record_content_string = parse_varlen_string parse_uint16 input in
  let record_content = parse_record_content content_type record_content_string in
  in {
    content_type = content_type;
    record_version = record_version;
    record_content = record_content;
  }
```

Analyse Le défaut majeur de cette approche est qu'il est nécessaire d'écrire énormément de code. En revanche, il est possible de décrire avec autant de détails qu'on peut le souhaiter un format et d'ajouter toutes les vérifications nécessaires, tout en gardant un code efficace.

Lors de nos premières implémentations en Python, C++ et OCaml, c'est cette approche que nous avons employée, afin de maîtriser au mieux les détails, parfois complexes et mal spécifiés, des formats et protocoles étudiés. Une fois écrite une bibliothèque de fonctions de base, les types étaient définis et les fonctions de *parsing* écrites manuellement. Cette méthode fonctionne, mais ne passe pas à l'échelle. De plus, étant donnée la quantité de code qu'il est nécessaire d'écrire, cela augmente d'autant les risques d'erreurs d'implémentation (y compris les erreurs typographiques).

2.5. Méthode 4 : déconstruction en place (*match*)

Description Une autre approche a été mise en œuvre dans un projet OCaml, l'extension `bitstring`[12]. L'idée y est d'étendre, à l'aide d'un pré-processeur `camlp4`, la notion de *pattern matching* pour être capable de déconstruire une chaîne binaire en place, tout comme le permet nativement le langage Erlang. Le code précédent s'écrirait ainsi, en utilisant `bitstring` :

```
let parse_record_content content_type bits =
  bitmatch bits with
  | { alert_level : 8 : bigendian;
      alert_type  : 8 : bigendian }
    when content_type = 0x15 -> Alert (alert_level, alert_type)
  | ...
  | { raw : -1 : string } -> Unparsed raw

let parse_record bits =
  bitmatch bits with
  | { content_type : 8 : bigendian;
      tls_version : 16 : bigendian;
      record_length : 16 : bigendian;
      record_content : record_length*8 : bitstring } ->
    content_type, tls_version, (parse_record_content content_type record_content)
  | { _ } -> failwith "Invalid Record"
```

Analyse Cette approche permet d'étendre le langage OCaml pour extraire des champs de bits dans un flux binaire, mais l'expressivité offerte par cette extension nous a semblé limitée. Bien que certaines constructions soient théoriquement exprimables (champs de taille variable, utilisation de discriminants à l'aide de clauses `when`), `bitstring` ne permet pas facilement d'ajouter des contraintes sur les différents champs ou de supporter les formats non linéaires ; dans les deux cas, il est nécessaire d'écrire du code OCaml pour lier les différents blocs `bitmatch`, qui sont alors spécifiques et difficilement réutilisables. De plus, le code produit par la construction `bitmatch` se révèle beaucoup moins efficace que la méthode précédente.

Pour notre étude, cette méthode ne nous a pas semblé adaptée aux formats étudiés, riches en contraintes ad-hoc. De plus, nous souhaitions vivement une méthode fournissant du code succinct, modulaire et largement réutilisable. Pour ces raisons, nous n'avons pas mis en œuvre cette méthode.

2.6. Méthode 5 : génération automatique de code (Pré-processeur)

Description Afin de bénéficier de tous les avantages de la méthode 3 sans en payer le prix en lignes de code, il est possible d'utiliser de la génération de code automatique. Nous avons en effet constaté, après avoir écrit le code pour plusieurs structures, que des *design patterns* émergent. Il est donc envisageable d'automatiser le traitement des énumérations, des structures et des unions, à l'aide d'une description qui sera l'entrée d'un pré-processeur. Avec la syntaxe de Parsifal (voir section 3), l'exemple précédent s'écrit simplement :

```
enum tls_version (16, UnknownVal V_Unknown) =
  | 0x0002 -> SSLv2
```

```
  | 0x0300 -> SSLv3
  | 0x0301 -> TLSv1
  | 0x0302 -> TLSv1_1
  | 0x0303 -> TLSv1_2

enum tls_content_type (8, Exception) =
  | 0x14 -> CT_ChangeCipherSpec
  | 0x15 -> CT_Alert
  | 0x16 -> CT_Handshake
  | 0x17 -> CT_ApplicationData

enum tls_alert_level (8, Exception) = ...
enum tls_alert_type (8, UnknownVal AT_Unknown) = ...

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

union record_content (Unparsed_Record) =
  | CT_Alert -> Alert of tls_alert
  | ...

struct tls_record = {
  content_type : tls_content_type;
  record_version : tls_version;
  record_content : container[uint16] of record_content (content_type)
}
```

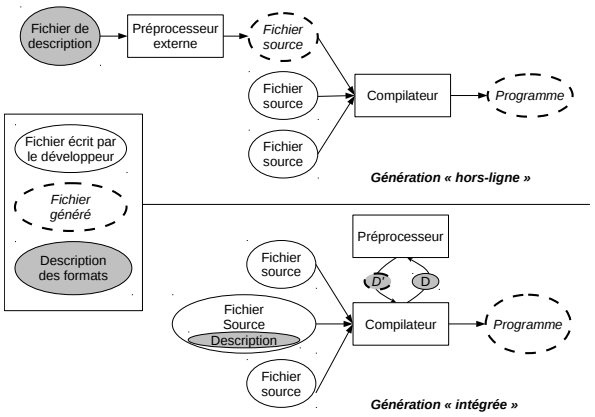

FIGURE 2 – Différence entre l'utilisation d'un pré-processeur externe et d'un pré-processeur intégré.

Analyse Cette méthode offre une description succincte (comme avec le DSL), tout en apportant les avantages de l'écriture manuelle (adéquation aux formats complexes, ajout de vérifications, efficacité).

Au-delà de la simple automatisation, il est également possible d'automatiser certains mécanismes, tels que la vérification de sommes de contrôle CRC. Pour faire cela, le pré-processeur peut générer la fonction *parse* associée au $\mathcal{PT}$ype `crc32_container of t` pour qu'elle réalise les opérations suivantes :

- *parsing* de l'élément `x` de type `t` à l'aide de `parse_t` ;
- calcul du CRC32 de la représentation binaire ;
- lecture du CRC32 depuis la source ;
- comparaison des deux sommes de contrôle (avec levée d'une exception en cas de différence) ;
- renvoi de la valeur construite `x`.

Il existe plusieurs projets reposant sur un préprocesseur pour générer du code à partir d'une description du format à analyser :

— dans le cadre du projet Bro, un système de détection d'intrusion (IDS), un langage, `binpac` [16], a été développé pour permettre de générer automatiquement du code C à partir de descriptions ;
— DataScript [5], un outil permettant de générer en Java des *parsers* pour des formats de fichiers binaires, et d'exprimer de nombreuses contraintes ;
— PacketTypes [14], un outil conçu pour décrire les protocoles réseau et générer du code C ;
— De même, une implémentation avait été réalisée à l'ANSSI, `FAceSL`, qui prenait des fichiers `.struct`, `.enum`, etc. pour générer le code OCaml correspondant.

Toutes ces implémentations fournissent de bons résultats, mais la génération de code *hors ligne* n'offre pas toute la souplesse nécessaire au traitement de certains formats de fichiers ou certains protocoles. En effet, il est nécessaire de générer le code à partir des descriptions, puis d'implémenter tout le reste dans le langage retenu.

Afin de surmonter ces limitations, nous avons décidé d'implémenter cette étape de génération à l'aide d'un pré-processeur intégré au processus de compilation : `camlp4`, pour le langage OCaml. La différence entre cette génération « hors-ligne » et la génération « intégrée » est illustrée à la figure 2.

2.7. Comparaison des méthodes

Voici un tableau récapitulatif des différentes solutions, vis-à-vis des propriétés attendues :

	cast	DSL	Manuelle	*match*	Pré-processeur
Langage représentatif	C	Python	tous	OCaml	tous
Implémentations issues de l'étude	-	OCaml	C++/Python	-	OCaml (2)
Adaptation aux formats complexes	--	+	++	-	++
Rapidité d'ecriture	+	++	--	+	++
Performances	++	-	+	-	+
Robustesse et possibilité d'implémenter des vérifications	-	+	++	+	++

3. Description de Parsifal

Parsifal est issu des besoins identifiés et de l'expérience acquise dans l'écriture de *parsers* pour des formats binaires. Il s'agit d'une implémentation générique de *parsers* reposant sur un pré-processeur `camlp4` et sur une bibliothèque auxiliaire. Comme l'ensemble des générateurs de code de la méthode 5, il remplit l'ensemble des besoins que nous avons identifiés, mais l'intégration de la génération de code à la compilation permet une plus grande flexibilité : il est possible de décrire la majorité des structures à l'aide du langage de description mis en place par le pré-processeur (mots clés `struct`, `union`, etc.) et de revenir à une écriture manuelle lorsque la situation l'exige.

3.1. Les $\mathcal{PT}$ypes

Le concept de base de Parsifal est la définition de « types enrichis », les $\mathcal{PT}$ypes, qui sont simplement des types OCaml quelconque pour lesquels un certain nombre de fonctions sont fournies. Ainsi, un $\mathcal{PT}$ype est défini par :

— un type OCaml `t` décrivant le contenu à *parser* ;
— des fonctions pour disséquer les objets correspondants depuis une chaîne de caractères (`parse_t`) ou depuis un flux Lwt [19] (`lwt_parse_t`) ;
— des fonctions pour exporter les objets sous forme binaire (`dump_t`) ou dans une représentation haut niveau utile à certaines fonctions d'affichage (`value_of_t`)[6].

6. L'ajout des fonctions `dump` et `value_of` est utile et quasiment gratuit, une fois la génération automatique de code

On peut distinguer trois sortes de $\mathcal{PT}$ypes :
— les $\mathcal{PT}$ypes de base (entiers, chaînes de caractère, listes, adresse IPv4, etc.), fournis par la bibliothèque `parsifal_core` ;
— les $\mathcal{PT}$ypes construits à partir de mots clés tels que `struct`, `union`, `enum`, etc. Pour ceux-ci, une description suffit au pré-processeur pour générer automatiquement le type OCaml et les fonctions correspondantes ;
— les $\mathcal{PT}$ypes personnalisés, c'est-à-dire des types OCaml classiques, suivis de la définition des fonctions attendues (au minimum `parse_t`, `dump_t` et `value_of_t`).

3.2. Quelques exemples de $\mathcal{PT}$ypes pour décrire le format TAR

Pour illustrer l'utilisation des constructions offertes par les mots clés, étudions un format de fichier relativement simple, TAR[7]. Il s'agit d'un format d'archive utilisé dans le monde Unix. Son rôle est d'agréger un ensemble de fichiers dans un seul gros fichier, qui pourra éventuellement être compressé ensuite. Une archive s'organise en une succession d'entrées, représentant chacune un fichier.

Enumérations

Chaque entrée est définie par le type de fichier qu'elle contient. Ce type est défini par un caractère, et peut prendre différentes valeurs, données dans le tableau ci-dessous :

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

Pour décrire une telle valeur, on utilisera le mot clé `enum`, qui crée des types qui ressemblent à l'`enum` du C, le typage statique en bonus. Ainsi, la déclaration suivante :

```
enum file_type (8, UnknownVal UnknownFileType) =
  | 0 -> NormalFile
  | 0x30 -> NormalFile
  | 0x31 -> HardLink
  | 0x32 -> SymbolicLink
    ...
  | 0x37 -> ContiguousFile
```

mènera à la génération du type suivant :

```
type file_type =
  | NormalFile | HardLink | SymbolicLink | ...
  | UnknownFileType of int
```

ainsi qu'à la génération de la fonction `val parse_file_type : string_input -> file_type`.

Les arguments de la déclaration `enum` indiquent respectivement la taille de l'entier représentant la valeur énumérée (`8` dans l'exemple) et le comportement à adopter en cas de valeur non reconnue (`UnknownVal` provoque la création d'un constructeur supplémentaire pour traiter ces cas, alors que le mot clé `Exception` aurait mené à la levée d'une exception).

mise en place. Ces fonctions ne seront pas décrits plus en détails dans ce document.

7. Dans Parsifal, le format TAR est présent dans les formats déjà décrits (`formats/tar.ml`), mais également sous la forme d'un tutoriel pas-à-pas (répertoire `tutorial/tar-steps/` du dépôt `https://github.com/ANSSI-FR/parsifal`).

Structures

Chaque entrée TAR commence par un en-tête dont les champs varient en fonction de la version du format utilisée ; le tableau ci-dessous décrit deux de ces versions : le format original (colonne TAR) et une variante plus récente (colonne `ustar`).

Offset en octets	Longueur en octets	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

On notera que l'énumération `file_type` décrite précédemment est utilisée pour le champ à l'octet 156. De plus, l'en-tête est contenu dans un « bloc » de 512 octets. Pour définir l'ensemble des champs présents dans un en-tête TAR, il suffit d'utiliser le mot clé `struct`[8] :

```
struct tar_header = {
  file_name : string(100);
  ...
  file_type : file_type;
  linked_file : string(100);
  ustar_magic : magic("ustar\x0000");
  ...
  filename_prefix : string(155);
  hdr_padding : binstring(12);
}
```

Unions

Afin de pouvoir traiter les différentes versions du format, on peut éclater l'enregistrement en deux : la partie commune (les champs avant `ustar_magic`) et la partie spécifique (potentiellement composée uniquement de caractères nuls pour le format TAR original).

Pour cela, il faut *parser* les premiers champs, lire le marqueur, et traiter le reste de l'en-tête en fonction de ce marqueur. Le mot clé correspondant à cette démarche dans Parsifal est `union`, qui va reposer sur un discriminant pour déterminer la manière de lire la suite :

```
struct ustar_header = {
  owner_user : string(32);
  owner_group : string(32);
  ...
  filename_prefix : string(155);
  hdr_padding : binstring(12);
}
```

8. En plus de $\mathcal{PT}$ypes basiques tels que les chaînes de caractères et de `file_type`, la structure utilise le $\mathcal{PT}$ype prédéfini `magic` (qui n'a rien à voir avec `Obj.magic`) pour définir un marqueur attendu.

```
union additional_header (NoMoreHeader of binstring(247)) =
  | "ustar\x000" -> UStarHeader of ustar_header
  | ... (* other versions of TAR headers *)

struct tar_header = {
  file_name    : string(100);
  ...
  file_type    : file_type;
  linked_file  : string(100);
  hdr_version  : string(8);
  hdr_extra    : additional_header (hdr_version);
}
```

Si `hdr_version` correspond au marqueur `ustar`, `hdr_extra` sera analysé comme une structure `ustar_header`. Dans le cas contraire, `hdr_extra` contiendra simplement une chaîne de 247 caractères, pour compléter le bloc de 512 octets.

Un $\mathcal{PT}$ype personnalisé

TAR présente une particularité quant au stockage des valeurs entières : celles-ci sont stockées sous la forme d'une chaîne de caractères représentant la valeur numérique en octal [9].

Dans les exemples ci-dessus, nous avons utilisé des chaînes de caractères classiques pour décrire de tels champs entiers, ce qui n'est pas satisfaisant, puisque la valeur réellement attendue est un entier.

Pour améliorer la situation, il suffit simplement de définir un nouveau type OCaml, ainsi que les fonctions associées. Nous disposerons alors d'un $\mathcal{PT}$ype utilisable à la place des chaînes de caractères brutes. Le code suivant décrit un nouveau type, `tar_numstring` pour prendre en compte ce format particulier :

```
type tar_numstring = int

let parse_tar_numstring len input =
  let octal_value = parse_string (len - 1) input in
  drop_bytes 1 input;
  try int_of_string ("0o" ^ octal_value)
  with _ -> raise (ParsingException (CustomException "int_of_string", _h_of_si input))

let dump_tar_numstring len buf v =
  bprintf buf "%*.*o\x00" len len v

let value_of_tar_numstring v = VSimpleInt v
```

Ensuite, il est possible de remplacer les champs de type `string(n)` par un champ de type `tar_numtring(8)`.

3.3. Présentations de fonctionnalités avancées

Les exemples des sections précédentes sont relativement simples, mais Parsifal offre à travers sa bibliothèque standard des fonctionnalités plus puissantes, toujours dans le but d'exprimer simplement des constructions complexes.

$\mathcal{PT}$ypes paramétrés Il est parfois nécessaire de transmettre des informations aux fonctions `parse`. Nous l'avons déjà vu dans le cas du discriminant utilisé pour une union, mais il peut également être nécessaire de transmettre en argument une longueur ou un contexte plus général. Bien que cela

9. La chaîne de caractères en question doit se terminer par un caractère nul ou un espace.

ne soit pas décrit en détail dans le présent document, Parsifal offre la possibilité de transmettre ce type d'information simplement. Un exemple de tel contexte est l'analyse du message TLS `ServerKeyExchange`, dont le contenu dépend de la suite cryptographique choisie par le serveur dans un autre message (`ServerHello`).

ASN.1 Dans le cadre du protocole TLS, il a été nécessaire d'analyser des certificats X.509, qui reposent sur l'encodage DER de l'ASN.1. Parsifal offre des constructions pour décrire des structures et unions ASN.1, ainsi que les $\mathcal{PT}$ypes de base. L'ASN.1 ne nécessite pas en soi de traitement spécifique, mais l'ajout de mots clés permet de libérer le développeur de l'analyse de l'en-tête DER (longueur et type), pour qu'il se concentre sur le contenu de la séquence.

Conteneurs Parsifal offre également une abstraction utile pour décrire des transformations à réaliser lors de la lecture de l'entrée binaire. Par exemple, le format PNG permet d'inclure des champs de commentaires compressés. Ils peuvent être exprimés simplement dans Parsifal à l'aide du code suivant [10] :

```
struct compressed_text = {
  key_word : cstring;
  compression_method : uint8;
  text : zlib_container of string;
}
```

La fonction `parse` générée se chargera de décompresser le texte à la lecture. Parmi les conteneurs déjà développés, on trouve la compression, l'encodage base64, la vérification de CRC et des conteneurs cryptographiques (PKCS#1 et PKCS#7). Tout comme les $\mathcal{PT}$ypes, les conteneurs peuvent être développés en OCaml en définissant un nouveau type et ses fonctions associées.

Champs de bits Afin de pouvoir décrire des champs tenant sur un nombre de bits non multiple de 8 (i.e. non alignés sur des octets), des $\mathcal{PT}$ypes spécifiques ont été implémentés [11], et les énumérations ont été adaptées en conséquence. Par exemple, les messages du protocole DNS utilisent ce type de champs :

```
enum opcode (4, UnknownVal UnknownOpcode) =
  | 0 -> StandardQuery
  | 1 -> InverseQuery
  | 2 -> ServerStatusRequest

struct dns_flags = {  (* champ de bits sur 16 bits *)
  qr : bit_bool;
  opcode : opcode;    (* énumération sur 4 bits *)
  (* ... *)
  rcode : bit_int[4];
}
```

Gestion fine du flot de contrôle du *parser* Afin de pouvoir accélérer le traitement de certains fichiers, il est possible de ne pas *parser* en détails certains champs. En effet, lorsqu'on s'intéresse uniquement aux paramètres cryptographiques négociés dans une session TLS, il n'est pas nécessaire de disséquer en détails les certificats.

De même, Parsifal permet d'introduire dans les descriptions de structures des champs fictifs (`parse_checkpoint` et `parse_field`) pour instrumenter le *parsing*, par exemple pour faire certaines

10. `cstring` désigne simplement une chaîne de caractères se terminant par un caractère nul.
11. De plus, les champs binaires peuvent se lire de la gauche vers la droite (en commençant par les bits de poids fort) ou de la droite vers la gauche (en utilisant le préfixe `rtol_`). En effet, si la première convention est largement répandue, la seconde est utilisée par le format de compression DEFLATE [8].

vérifications. Un autre cas d'instrumentation est pour se déplacer dans le fichier à analyser ; en effet, certains formats (entre autres PE [12] et EXIF [13]) nécessitent d'utiliser des *offsets* explicites pour décoder leur contenu.

4. Résultats et enseignements

4.1. Retours d'expérience généraux

Par rapport à de nombreuses autres approches testées, Parsifal a montré qu'il avait toutes les propriétés attendues :

— adaptation à la description de structures complexes ;
— performances du code produit ;
— grande expressivité du langage permettant dans la majorité des cas un code succinct ;
— possibilité d'écrire du code à la main lorsque la situation l'exige ;
— robustesse.

Il est important de noter que la robustesse découle essentiellement de l'utilisation d'OCaml (en particulier grâce à son typage fort et à sa gestion mémoire automatique) et de l'automatisation des tâches ingrates (une fois le préprocesseur mis au point, il y a beaucoup moins de code à écrire).

De plus, si une partie des tâches automatisées concernent les détails de bas niveau (vérification de la longueur exacte des champs, correspondance des étiquettes ASN.1 attendues, etc.), une autre partie de l'automatisation concerne des vérifications de plus haut niveau (vérification d'un CRC ou d'une signature cryptographique).

En revanche, si Parsifal permet de vérifier et générer des flux conformes, il n'est pas adapté au *fuzzing* (c'est-à-dire à la génération de trames ou fichiers déviant de la spécification initiale), ce qui représente une prise de parti différente d'autres outils (comme Scapy et Hachoir).

4.2. Formats supportés

Depuis 2 ans, plusieurs formats ont été décrits à l'aide de Parsifal. En voici la liste, en insistant sur les difficultés rencontrées.

Travaux initiaux sur SSL/TLS L'ensemble des messages du protocole TLS, et une partie des messages du protocole SSLv2 ont été décrits. Il en est de même des certificats X.509. Les principales difficultés rencontrées sont :

— le nombre important de structures/unions/énumérations nécessaires (en particulier, pour analyser correctement une trace PCAP [14] contenant des messages TLS, il est nécessaire de gérer correctement l'accumulation des formats PCAP, TCP/IP, TLS, X.509 et ASN.1) ;
— la nécessité de conserver un contexte pour analyser correctement les messages qui dépendent de paramètres négociés auparavant.

TAR : le premier tutoriel Afin de démontrer que Parsifal était adapté à d'autres sujets d'étude que SSL/TLS, un tutoriel a été écrit pour analyser le format TAR. En quelques dizaines de lignes, il est possible de s'approprier le fonctionnement de l'outil. Des exemples issus de ce tutoriel ont été présentés dans la section 3.2.

12. Le format PE (*Portable Executable*) est le format d'exécutables utilisé par Windows et UEFI.
13. Le format EXIF (*Exchangeable image file format*) est utilisé dans les images TIFF et JPEG pour stocker des métadonnées.
14. PCAP est un standard répandu pour la sauvegarde de captures réseau.

DNS : un défi interne Le protocole DNS (*Domain Name Server* [15]), qui permet d'associer à des noms de domaines diverses informations (dont les adresses IP), utilise de la compression dans ses messages : si un nom de domaine a déjà été (même partiellement) écrit dans le message, il est possible d'y faire référence avec un pointeur vers l'offset dans le message. De plus, DNS fut le premier format à nécessiter l'utilisation de champs de bits. Ces particularités ont été implémentées dans Parsifal, et l'implémentation résultante permet d'écrire un client DNS fonctionnel en moins de 200 lignes de code.

Travaux autour de l'UEFI Le laboratoire architectures matérielles et logicielles de l'ANSSI s'est intéressé à l'UEFI [15]. Cette fois, le défi venait de la présence d'un format (PE) dont l'analyse ne peut être linéaire. Plusieurs outils ont été développés pour décortiquer les exécutables UEFI, dont certains à l'aide de Parsifal. Ces travaux ont fait l'objet d'une publication [7].

Autres formats Enfin, diverses études ont mené à l'écriture d'outils pour manipuler des fichiers images (PNG et JPEG), des traces PCAP contenant des paquets IP/TCP/UDP, des archives contenant des annonces BGP, ou encore des messages du protocole Kerberos.

4.3. Limitations actuelles et perspectives

À ce jour, Parsifal ne sait pas gérer la définition récursive de $\mathcal{PT}$ypes à l'aide des constructions `struct` et `union`. Bien que cela soit rare dans les formats binaires étudiés, des formats comme DEX (le *bytecode* utilisé par les plateformes Androïd) en font usage. De même, il serait plus naturel de décrire les formats EXIF ou PKCS#7 à l'aide de définitions récursives (par exemple, la miniature EXIF d'une image JPEG est elle-même une image JPEG).

Ensuite, Parsifal n'offre que peu d'outils pour manipuler les valeurs construites. Si cela suffit dans de nombreux cas, il existe des formats complexes consistant en des structures imbriquées à des niveaux de profondeur importants. Des travaux sont donc envisagés pour permettre une manipulation naturelle des valeurs construites.

Nous souhaiterions également expérimenter Parsifal pour écrire des outils à vocation un peu plus opérationnelle, avec des *proxy* filtrants ou normalisants, par exemple pour dépolluer les images et documents PDF téléchargés depuis internet. L'intérêt de cette normalisation serait d'assainir et de simplifier la structure de l'image ou du document pour réduire les risques d'exploitation d'une vulnérabilité dans un logiciel en aval.

Enfin, une piste à plus long terme serait d'écrire et d'animer complètement le protocole de négociation TLS (à ce jour, seule une animation rudimentaire a été développée) pour fournir une pile TLS de confiance.

5. Conclusion

Pour développer une meilleure compréhension des protocoles, l'ANSSI a développé plusieurs outils de dissection de protocoles, dont Parsifal. Ce dernier est une implémentation générique de *parsers* reposant sur un pré-processeur `camlp4` et sur une bibliothèque auxiliaire. Parsifal permet de générer des dissecteurs réunissant toutes les propriétés recherchées (concision, rapidité de développement, garanties fortes sur la robustesse et performances des programmes). L'utilisation de Parsifal s'est montrée très efficace, y compris pour des personnes ne connaissant pas OCaml au départ [16].

15. L'UEFI (*Unified Extensible Firmware Interface*) est le remplaçant du BIOS (*Basic Input Output System*), le système de démarrage des PC.

16. Ainsi, nous avons pu constater qu'un stagiaire non formé à la programmation fonctionnelle pouvait appréhender l'outil, et malgré une prise en main plus longue que dans le cas du langage C, les programmes obtenus avec OCaml (et Parsifal) étaient plus robustes et plus courts.

Concrètement, des outils d'analyse ont été réalisés pour étudier les protocoles TLS et Kerberos, mais aussi des formats de fichiers tels que PE, PNG, et JPEG. Concernant le format d'image PNG, des programmes de normalisation (ou de dépollution) ont également été implémentés.

Enfin, le code source est aujourd'hui disponible sur GitHub (`https://github.com/ANSSI-FR/parsifal`) et contient trois tutoriels pas-à-pas pour décrire les formats TAR, DNS et PNG.

Remerciements

Je souhaiterais remercier Eric, Thérèse et Céline pour leurs relectures attentives, ainsi que Pierre, Thomas et Anthony pour leurs contributions à Parsifal.

Références

[1] ANSSI. Langages et sécurité - généralités et cas des langages fonctionnels. In *JFLA 2013*.

[2] ANSSI. Sécurité et langage Java. `http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/securite-et-langage-java.html`, 2010.

[3] ANSSI. LaFoSec : Sécurité et langages fonctionnels. `http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/lafosec-securite-et-langages-fonctionnels.html`, 2013.

[4] 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.

[5] G. Back. Datascript - A Specification and Scripting Language for Binary Data. In *Generative Programming and Component Engineering*. Springer, 2002.

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

[7] P. Chifflier. UEFI et bootkits PCI : le danger vient d'en bas. In *SSTIC*, 2013.

[8] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

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

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

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

[12] R. Jones. bitstring. `http://code.google.com/p/bitstring/`, 2003-2012.

[13] O. Levillain, A. Ébalard, H. Debar, and B. Morin. One Year of SSL Measurement. In *ACSAC*, 2012.

[14] P. J. McCann and S. Chandra. Packet types : Abstract specifications of network protocol messages. In *SIGCOMM*, pages 321–333, 2000.

[15] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.

[16] Ruoming Pang and Robin Sommer. binpac : A yacc for Writing Application Protocol Parsers. In *Internet Measurement Conference*, 2006.

[17] Saferiver. Le projet LaFoSec. In *JFLA 2013*.

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

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