GASP: a Generic Approach to Secure network Protocols

Olivier Levillain

May 13th 2020
Agenda

Introduction

The Need for Robust Parsers

A Platform for Binary Parser Generators

Animating Protocols

Fuzzing implementations

Next steps
Agenda

Introduction

The Need for Robust Parsers

A Platform for Binary Parser Generators

Animating Protocols

Fuzzing implementations

Next steps
Introduction

Project Outline

GASP, a Generic Approach to Secure Protocols

- Project funded by the ANR 2019 call (ANR Jeune)
- 4 ans (2019-10-01 – 2023-09-30)

Three main research directions

- Network protocol observation in the field
- Protocol description to derive reference implementation
- Tests on existing implementations using a grey- or whitebox approach

Ressources

- 1 PhD student (ATR) + 3 interns (incl. SN)
- 20 k€ for servers/laptops
- 25 k€ for travel/conferences
Partners

Télécom SudParis
► Olivier Levillain, principal investigator
► Aina Toky Rasoamanana, PhD student

ANSSI (software security lab)
► Arnaud Fontaine
► Aurélien Deharbe

Collegues from Rennes
► Georges Bossert (Sekoia), pylstar developer
► Guillaume Hiet (CentraleSupélec)

Other people involved
► Karthik Bhargavan (Inria Paris, Prosecco)
► Pascal Lafourcade (UCA)
► Graham Steel (Cryptosense)
## Deliverables and tasks (1/2)

<table>
<thead>
<tr>
<th>WP0</th>
<th>Project management and dissemination</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0.*</td>
<td>Yearly progress reports</td>
</tr>
<tr>
<td>WP1</td>
<td>Network protocol observation in the field</td>
</tr>
<tr>
<td>WP1.1</td>
<td>Specification of a message description language</td>
</tr>
<tr>
<td>WP1.2</td>
<td>Development of compilers to derive parsers</td>
</tr>
<tr>
<td>WP1.3</td>
<td>Measurement campaigns</td>
</tr>
<tr>
<td>D1.1</td>
<td>Intermediate report on the message language and compilers</td>
</tr>
<tr>
<td>D1.2</td>
<td>Final report on the message language and compilers</td>
</tr>
<tr>
<td>D1.3</td>
<td>Campaigns results (tools, data and analyses)</td>
</tr>
<tr>
<td>WP2</td>
<td>Protocol description to derive reference implementations</td>
</tr>
<tr>
<td>WP2.1</td>
<td>Specification of a protocol description languages</td>
</tr>
<tr>
<td>WP2.2</td>
<td>Development of compilers to derive reference implementations</td>
</tr>
<tr>
<td>D2.1</td>
<td>Intermediate report on the languages and compilers</td>
</tr>
<tr>
<td>D2.2</td>
<td>Final report on the languages and compilers</td>
</tr>
<tr>
<td>WP3</td>
<td>Tests on existing implementations using a grey- or whitebox approach</td>
</tr>
<tr>
<td>WP3.1</td>
<td>Test tools derived from the description languages</td>
</tr>
<tr>
<td>WP3.2</td>
<td>Program introspection to explore implementation behaviour</td>
</tr>
<tr>
<td>D3.1</td>
<td>Intermediate report on test tools</td>
</tr>
<tr>
<td>D3.2</td>
<td>Final report on test tools</td>
</tr>
<tr>
<td>D3.3</td>
<td>Report on implementation introspection</td>
</tr>
</tbody>
</table>
Deliverables and tasks (2/2)
Agenda

Introduction

The Need for Robust Parsers

A Platform for Binary Parser Generators

Animating Protocols

Fuzzing implementations

Next steps
Network protocols and file formats

- To understand a specification, you should try and implement it
- Often, the devil in the detail
  - how to encode integers in ASN.1, tar files or protobuf
  - the direction to fill in bit fields
  - fuzzy specifications
- Binary parsers are a basic block for a lot of programs
- They are often a fragile part of the software (look at CVEs for Wireshark for example)
Where it all began: SSL/TLS campaigns

- Analysis of SSL/TLS connections in the wild (ACSAC 2012)
  - for each 443/tcp open port, we record the answer to a given stimulus
  - 200 GB of raw data per stimulus

- Problems to handle and dissect these data
  - TLS is composed of complex structured messages
  - data can be corrupted (in many ways)
  - 443/tcp can host other protocols (usually HTTP or SSH)
  - more subtle errors in messages
Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites: \texttt{AES128-SHA} et \texttt{ECDH-ECDSA-AES128-SHA}?
Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites: 
**AES128-SHA** et **ECDH-ECDSA-AES128-SHA**?

A **AES128-SHA**
Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites:

A. AES128-SHA  
B. ECDH-ECDSA-AES128-SHA

A. AES128-SHA  
B. ECDH-ECDSA-AES128-SHA
Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites: **AES128-SHA** et **ECDH-ECDSA-AES128-SHA** ?

A AES128-SHA

B ECDH-ECDSA-AES128-SHA

C an alert
Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites: AES128-SHA et ECDH-ECDSA-AES128-SHA?

A. AES128-SHA
B. ECDH-ECDSA-AES128-SHA
C. an alert
D. something else (RC4_MD5)
Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites:

A AES128-SHA

B ECDH-ECDSA-AES128-SHA

C an alert

D something else (RC4_MD5)

Actually, it is easy to explain
Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites: **AES128-SHA** et **ECDH-ECDSA-AES128-SHA**?

A. **AES128-SHA** (0x002f)

B. **ECDH-ECDSA-AES128-SHA**

C. an alert

D. something else (**RC4_MD5**) (0x0005)

Actually, it is easy to explain

- a ciphersuite is represented by a 16-bit integer
- for almost a decade, all suites had their first byte equal to 00
Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites: **AES128-SHA** et **ECDH-ECDSA-AES128-SHA**?

A. **AES128-SHA** (0x002f)
B. **ECDH-ECDSA-AES128-SHA** (0xc005)
C. an alert
D. something else (**RC4_MD5**) (0x0005)

Actually, it is easy to explain

- a ciphersuite is represented by a 16-bit integer
- for almost a decade, all suites had their first byte equal to 00
The Need for Robust Parsers

Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites: AES128-SHA et ECDH-ECDSA-AES128-SHA?

A. AES128-SHA (0x002f)
B. ECDH-ECDSA-AES128-SHA (0xc005)
C. an alert
D. something else (RC4_MD5) (0x0005)

Actually, it is easy to explain

- a ciphersuite is represented by a 16-bit integer
- for almost a decade, all suites had their first byte equal to 00
- why bother to inspect this byte?
The Need for Robust Parsers

Home-made SSL/TLS stacks

What should a client expect when they propose the following ciphersuites: **AES128-SHA** and **ECDH-ECDSA-AES128-SHA**?

- A **AES128-SHA**
- B **ECDH-ECDSA-AES128-SHA**
- C an alert
- D something else (**RC4_MD5**)
- E an otherwise correct message, where the field is *missing*
Parsifal, a brochure

- A tool to write parsers from **concise** descriptions
- **Efficiency** of the compiled programs
- **Robustness** of the developed tools
- Development methodology adapted to an **incremental** approach to produce flexible parsers
The Need for Robust Parsers

Parsifal, a brochure

- A tool to write parsers from **concise** descriptions
- **Efficiency** of the compiled programs
- **Robustness** of the developed tools
- Development methodology adapted to an **incremental** approach to produce flexible parsers

- Parsifal also allows to dump/unparse the objects
- Example: a simple DNS client in 200 lines
The Need for Robust Parsers

Parsifal base concept : the PType

The objects to analyse are described using PTypes

- an OCaml type
- a parse function
- a dump function

Different sorts of PTypes

- base PTypes (uint, binstring, etc.)
- Parsifal constructions using keywords (enum, struct, etc.)
- hand-written PTypes
Exemple : structure d’une image PNG (1/3)

```
struct png_file = {
    png_magic : magic( "\x89\x50\x4e\x47\x0d\x0a\x1a\x0a" );
    png_content : bstring;
}
```
Exemple : structure d’une image PNG (2/3)

```c
struct png_chunk = {
    chunk_size : uint32;
    chunk_type : string(4);
    data : binstring(chunk_size);
    crc : uint32;
}
```
Exemple : structure d’une image PNG (2/3)

```c
struct png_chunk = {
  chunk_size : uint32;
  chunk_type : string(4);
  data : binstring(chunk_size);
  crc : uint32;
};

struct png_file = {
  png_magic : magic("\x89\x50\x4e\x47\x0d\x0a\x1a\x0a");
  chunks : list of png_chunk;
};
```
Exemple : structure d’une image PNG (3/3)

```c
struct image_header = {
    ...
}

union chunk_content [enrich] (UnparsedChunkContent) =
    | "IHDR"  -> ImageHeader of image_header
    | "IDAT"  -> ImageData of binstring
    | "IEND"  -> ImageEnd
    | "PLTE"  -> ImagePalette of list of array(3) of uint8
```
Exemple : structure d’une image PNG (3/3)

```c
struct image_header = {
    ...
}

union chunk_content [enrich] (UnparsedChunkContent) =
| "IHDR" -> ImageHeader of image_header
| "IDAT" -> ImageData of binstring
| "IEND" -> ImageEnd
| "PLTE" -> ImagePalette of list of array(3) of uint8

struct png_chunk = {
    chunk_size : uint32;
    chunk_type : string(4);
    data : container(chunk_size) of chunk_content(chunk_type);
    crc : uint32;
}
```
Interlude: integer representation

How to represent 1034 (0b010000001010, 0x40a) and 10 (0b1010, 0xa)?
Interlude : integer representation

How to represent 1034 ($0b010000001010$, 0x40a) and 10 ($0b1010$, 0xa) as an ASN.1 integer (DER)?

▶ as the object length in ASN.1 (DER)?

▶ as a tag in ASN.1 (DER)

▶ as the file size (or any integer) in TAR?

▶ the string "00000002012" (octal representation)
Interlude: integer representation

How to represent 1034 (0b010000001010, 0x40a) and 10 (0b1010, 0xa)?

- as an ASN.1 integer (DER)?
  - 0x02 0x04 0x0a (len=2)
  - 0x01 0x0a (len=1)
Interlude: integer representation

How to represent $1034$ ($0b010000001010$, $0x40a$) and $10$ ($0b1010$, $0xa$)?

- as an ASN.1 integer (DER)?
  - $0x02$ $0x04$ $0xa$ ($\text{len}=2$)
  - $0x01$ $0xa$ ($\text{len}=1$)
- as the object length in ASN.1 (DER)?
Interlude: integer representation

How to represent 1034 (0b010000001010, 0x40a) and 10 (0b1010, 0xa)?

▶ as an ASN.1 integer (DER)?
  ▶ 0x02 0x04 0x0a (len=2)
  ▶ 0x01 0x0a (len=1)

▶ as the object length in ASN.1 (DER)?
  ▶ 0x82 0x04 0x0a (long format, len=2)
  ▶ 0x0a (short format, implicit len=1)
Interlude: integer representation

How to represent 1034 (0b010000001010, 0x40a) and 10 (0b1010, 0xa)?
  ▶ as an ASN.1 integer (DER)?
    ▶ 0x02 0x04 0x0a (len=2)
    ▶ 0x01 0x0a (len=1)
  ▶ as the object length in ASN.1 (DER)?
    ▶ 0x82 0x04 0x0a (long format, len=2)
    ▶ 0x0a (short format, implicit len=1)
  ▶ as a tag in ASN.1 (DER)
Interlude : integer representation

How to represent 1034 (0b010000001010, 0x40a) and 10 (0b1010, 0xa)?

▶ as an ASN.1 integer (DER)?
  ▶ 0x02 0x04 0x0a (len=2)
  ▶ 0x01 0x0a (len=1)

▶ as the object length in ASN.1 (DER)?
  ▶ 0x82 0x04 0x0a (long format, len=2)
  ▶ 0x0a (short format, implicit len=1)

▶ as a tag in ASN.1 (DER)
  ▶ 0b11111 0b10001000 0b00001010 (long format, last 7-bit chunk signaled by msb)
  ▶ 0b01010 (short format, implicit len=1)
Interlude: integer representation

How to represent $1034 \ (0b010000001010, \ 0x40a)$ and $10 \ (0b1010, \ 0xa)$?

- as an ASN.1 integer (DER)?
  - $0x02 \ 0x04 \ 0x0a \ (len=2)$
  - $0x01 \ 0x0a \ (len=1)$

- as the object length in ASN.1 (DER)?
  - $0x82 \ 0x04 \ 0x0a \ (long \ format, \ len=2)$
  - $0x0a \ (short \ format, \ implicit \ len=1)$

- as a tag in ASN.1 (DER)
  - $0b11111 \ 0b10001000 \ 0b00001010 \ (long \ format, \ last \ 7-bit \ chunk \ signaled \ by \ msb)$
  - $0b01010 \ (short \ format, \ implicit \ len=1)$

- as the file size (or any integer) in TAR?
  - the string "00000002012" (octal representation)
  - the string "00000000012"
Interlude: integer representation

How to represent \(1034\) (\(0b010000001010, 0x40a\)) and \(10\) (\(0b1010, 0xa\))?

- as an ASN.1 integer (DER)?
  - \(0x02\) \(0x04\) \(0x0a\) (len=2)
  - \(0x01\) \(0x0a\) (len=1)

- as the object length in ASN.1 (DER)?
  - \(0x82\) \(0x04\) \(0x0a\) (long format, len=2)
  - \(0x0a\) (short format, implicit len=1)

- as a tag in ASN.1 (DER)
  - \(0b11111\) \(0b10001000\) \(0b00001010\) (long format, last 7-bit chunk signaled by msb)
  - \(0b01010\) (short format, implicit len=1)

- as the file size (or any integer) in TAR?
  - the string "00000002012"
  - the string "00000000012"
Interlude: integer representation

How to represent 1034 (0b010000001010, 0x40a) and 10 (0b1010, 0xa)?

- as an ASN.1 integer (DER)?
  - 0x02 0x04 0x0a (len=2)
  - 0x01 0x0a (len=1)
- as the object length in ASN.1 (DER)?
  - 0x82 0x04 0x0a (long format, len=2)
  - 0x0a (short format, implicit len=1)
- as a tag in ASN.1 (DER)
  - 0b11111 0b10001000 0b00001010 (long format, last 7-bit chunk signaled by msb)
  - 0b01010 (short format, implicit len=1)
- as the file size (or any integer) in TAR?
  - the string "00000002012" (octal representation)
  - the string "00000000012"
Agenda

Introduction

The Need for Robust Parsers

A Platform for Binary Parser Generators

Animating Protocols

Fuzzing implementations

Next steps
Parsifal Limitations

Parsifal drawbacks

- OCaml adherence...
- and in particular to camlp4
- rather unsound handling of non linear constructions
- lack of a cool interpreter to help discovery

New ideas

- look ar other languages, e.g. Rust (and its Nom library)
- enrich the DSL (domain-specific language) to reason on PTypes
- better handle constraints on fields
- better isolate parsing from semantic interpretation
Other Tools and Languages

A lot of competitors, including

- Hammer (C)
- Scapy (Python)
- Hachoir (Python)
- *Parsifal* (OCaml)
- Netzob (Python)
- Nail (C)
- Nom (Rust)
- RecordFlux (Ada)
- Everparse (F*)
Other Tools and Languages

A lot of competitors, including

- Hammer (C)
- Scapy (Python)
- Hachoir (Python)
- *Parsifal* (OCaml)
- Netzob (Python)
- Nail (C)
- Nom (Rust)
- RecordFlux (Ada)
- Everparse (F*)

How to compare these tools?

- expressiveness
- robustness
- simplicity
Our Platform

This is a very young Work-In-Progress, to test tools on specifications, with regards to several properties.
Our Platform

This is a very young Work-In-Progress, to test tools on specifications, with regards to several properties.

Tools

- Hammer
- Nail
- Nom
- Parsifal
Our Platform

This is a very young Work-In-Progress, to test tools on specifications, with regards to several properties.

Tools
- Hammer
- Nail
- Nom
- Parsifal

Specifications
- trivial structures (to document how to handle basic fields)
- DNS
- PNG (and Mini-PNG)
Our Platform

This is a very young Work-In-Progress, to test tools on specifications, with regards to several properties.

Tools
- Hammer
- Nail
- Nom
- Parsifal

Specifications
- trivial structures (to document how to handle basic fields)
- DNS
- PNG (and Mini-PNG)

Properties
- sample validation
- parsing speed (not implemented yet)
- robustness (not implemented yet)
DNS on the Platform (1/2)

Various samples:

- valid requests and answers...
- including modern features
- truncated messages
- corrupted messages with invalid pointers
DNS on the Platform (1/2)

Various samples:

- valid requests and answers...
- including modern features
- truncated messages
- corrupted messages with invalid pointers

<table>
<thead>
<tr>
<th>Tool</th>
<th>Lines</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hammer</td>
<td>254</td>
<td>Limited fields</td>
</tr>
<tr>
<td>Nail</td>
<td>141</td>
<td>Compression, Zone description</td>
</tr>
<tr>
<td>Nom</td>
<td>88</td>
<td>Basic message structure</td>
</tr>
<tr>
<td>Parsifal</td>
<td>234</td>
<td>Various message types, Compression</td>
</tr>
</tbody>
</table>
DNS on the Platform (2/2)

Lessons learned from the behaviours of the different tools

- original and current specifications are in conflict (reserved field)
- DNS Extensions are not recognized by some implementations
- some field values are hardcoded in the proposed specs
- DNS compression is not always implemented, and usually requires specific hand-written code
DNS on the Platform (2/2)

Lessons learned from the behaviours of the different tools

▶ original and current specifications are in conflict (reserved field)
▶ DNS Extensions are not recognized by some implementations
▶ some field values are hardcoded in the proposed specs
▶ DNS compression is not always implemented, and usually requires specific hand-written code

Sebastien Naud, intern at TSP, is currently working on DNS and Nail.
▶ Short presentation at R3S Seminar next week (May 20th)
One important goal for GASP

We would like to propose a new DSL (domain-specific language) that would take the best of everything if possible

- concision
- expressiveness
- language-agnostic

![Diagram: How Standards Proliferate](https://xkcd.com/927/)

The approach would be to design a language and to implement compilers towards interesting programming languages or other DSLs
A new vision for structs

```plaintext
struct png_chunk = {
    chunk_size : uint32;
    chunk_type : string (4);
    chunk_data : chunk_content;
    chunk_crc : uint32;
}

constraints {
    chunk_size = len(chunk_data);
    chunk_crc = crc32(chunk_type ^ chunk_data);
    chunk_type = discriminant(chunk_data)
}
```
A new vision for structs

```c
struct png_chunk = {
    chunk_size : uint32;
    chunk_type : string (4);
    chunk_data : chunk_content;
    chunk_crc : uint32;
}
```

```c
constraints {
    chunk_size = len(chunk_data);
    chunk_crc = crc32(chunk_type ^ chunk_data);
    chunk_type = discriminant (chunk_data)
}
```

- We define functional relations useful for parsing and dumping
- To produce a valid png_chunk only requires the data field
  - chunk_data = ImageHeader ... implies that...
  - chunk_size is computable
  - chunk_type is "IHDR"
  - chunk_crc is computable
Agenda

Introduction

The Need for Robust Parsers

A Platform for Binary Parser Generators

Animating Protocols

Fuzzing implementations

Next steps
State machine description

Similarly to message formats, we would like a DSL to capture state machines and protocol contexts
State machine description

Similarly to message formats, we would like a DSL to capture state machines and protocol contexts.

Currently, very little animation done with Parsifal:

- picodig, a trivial DNS client
- simple TLS state machines
  - a decryption tool using SSLKEYLOG files
  - a proxy routing records depending on the first packets

More work is needed (WP2) before we can abstract out what is needed.
Agenda

Introduction

The Need for Robust Parsers

A Platform for Binary Parser Generators

Animating Protocols

Fuzzing implementations

Next steps
Principle of L* 

L* is an algorithm to infer automata

- original paper: Dana Anglui — *Learning Regular Sets from Queries and Countermeasures*, 1987
- initial scope is very limited since it requires to have a way to decide the equivalence with an ideal implementation
- approximations are possible to infer a state machine in a black box situation with reasonable precision
Application to protocol implementations

To interact with the implementation to test (as a black box), we need to

▶ concretize the messages to send
▶ abstract the received messages
▶ the algorithm will drive the request to explore the state machine

In practice, different kinds of received messages

▶ real message
▶ error
▶ time out
Some references about this approach

**TLS**
  - https://www.usenix.org/node/190893

**H2**
- Georges Bossert – *Comparaisons et attaques sur le protocole HTTP2* (SSTIC 2016)

**SSH**
- Fiterau-Brostean et al. – *Model Learning and Model Checking of SSH Implementations* (SPIN’17)
Example of a discovered flaw (1/2)

Observable state automata of the RSA BSAFE JAVA stack (version 6.1.1)

- 5 states clearly form the expected “happy flow”
- the 2 state is the error state
- Source: de Ruiters and Poll, Usenix Security 2015
Example of a discovered flaw (2/2)

Observable state automata of GNU TLS 3.3.8

- the automata contains 12 states
- states 8 to 10 form a shadow flow, a Heartbeat leading to a reset
- Source: de Ruiters and Poll, Usenix Security 2015
Ideas to improve and extend L*

Performance improvements

▶ timeout detections by introspection
▶ freeze/fork/restart to speed up the number of test cases
Ideas to improve and extend L*

Performance improvements
- timeout detections by introspection
- freeze/fork/restart to speed up the number of test cases

Alphabet extension
- use more detailed messages
- add corrupted/invalid messages
- take into account the time spent
- application: automatic detection of Bleichenbacher attacks in TLS implementations
Ideas to improve and extend L*  

Performance improvements  
▶ timeout detections by introspection  
▶ freeze/fork/restart to speed up the number of test cases  

Alphabet extension  
▶ use more detailed messages  
▶ add corrupted/invalid messages  
▶ take into account the time spent  
▶ application: automatic detection of Bleichenbacher attacks in TLS implementations  

More on this next week (R3S Seminar, May 20th), with a presentation by Aina Toky Rasoamanana, PhD student
Agenda

Introduction

The Need for Robust Parsers

A Platform for Binary Parser Generators

Animating Protocols

Fuzzing implementations

Next steps
Next steps (1/3)

Binary Parsers Platform

- stabilize the platform with 5-6 tools and several specs
- invite tool developers to join
- include performance tests
Next steps (1/3)

Binary Parsers Platform

- stabilize the platform with 5-6 tools and several specs
- invite tool developers to join
- include performance tests

L*

- better understand pylstar
- or implement a new version of L*?
- improve the performance with a grey-box approach
Use the message parsers to work on several ecosystems (network scans, implementation tests)

- TLS (as a benchmark)
- QUIC
- SSH
- H2
- ...

Next steps (2/3)
Next steps (3/3)

DSL to describe protocol messages

- Language design
- Compiler implementations
Next steps (3/3)

DSL to describe protocol messages
- Language design
- Compiler implementations

Protocol animation
- implement protocol stacks for different protocols
- abstract out a way to describe these implementations
- derive reference implementations
Questions?

Thank you for your attention

Do not hesitate to speak up if you are interested to contribute!
Backup slides
## Parsifal: implemented formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.509</td>
<td>rather complete description</td>
</tr>
<tr>
<td>SSL/TLS</td>
<td>most TLS &lt; 1.3 messages</td>
</tr>
<tr>
<td></td>
<td>rudimentary TLS 1.0 implementation</td>
</tr>
<tr>
<td>Kerberos</td>
<td>PKINIT messages</td>
</tr>
<tr>
<td>BGP/MRT</td>
<td>tool to extract the prefixes announced</td>
</tr>
<tr>
<td>DNS</td>
<td>tutorial + picodig</td>
</tr>
<tr>
<td>NTP</td>
<td>several messages</td>
</tr>
<tr>
<td>TAR</td>
<td>tutorial</td>
</tr>
<tr>
<td>PNG</td>
<td>tutorial</td>
</tr>
<tr>
<td>OpenPGP</td>
<td>packet structure</td>
</tr>
<tr>
<td>DVI</td>
<td>simple dissection</td>
</tr>
</tbody>
</table>