GASP: a Generic Approach to Secure network Protocols

Olivier Levillain

May 13th 2020

Agenda

Agenda

Introduction

The Need for Robust Parsers

A Platform for Binary Parser Generators

Animating Protocols

Fuzzing implementations

Next steps

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*

Ressourcess

- 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)

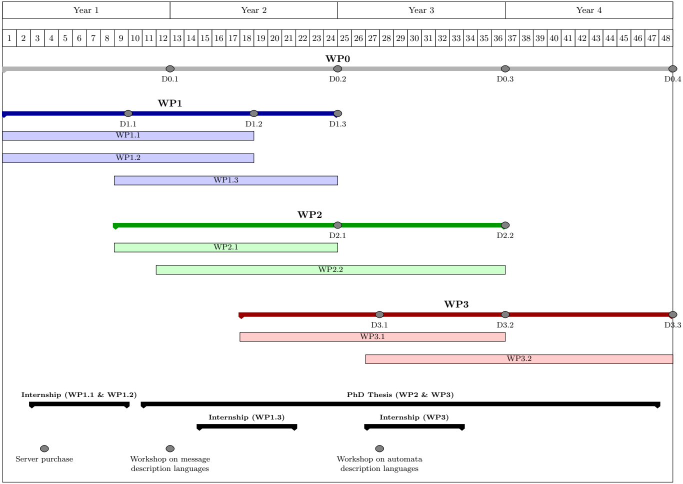
Other people involved

- Karthik Bhargavan (Inria Paris, Prosecco)
- Pascal Lafourcade (UCA)
- Graham Steel (Cryptosense)

Deliverables and tasks (1/2)

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

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 : `AES128-SHA` et `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 : `AES128-SHA` et `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 : `AES128-SHA` et `ECDH-ECDSA-AES128-SHA` ?

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`)

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

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 ?

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`)

E an otherwise correct message, where the field is *missing*

Parsifal, a brochure

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

Parsifal, a brochure

- A tool to write parsers from **concise** descriptions
- **Efficience** 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

Parsifal base concept : the PType

The objects to analyse are described using PTypes

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

Differentes 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 : binstring;
}
```

Exemple : structure d'une image PNG (2/3)

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

Exemple : structure d'une image PNG (2/3)

```
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)

```
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)

```
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) ?

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 0x0a` (len=2)
  - `0x01 0x0a` (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 ?

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 ($F^\star$)

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 ($F^\star$)

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

Tool	**Lines**	**Features**
Hammer	254	Limited fields
Nail	141	Compression, Zone description
Nom	88	Basic message structure
Parsifal	234	Various message types, Compression

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

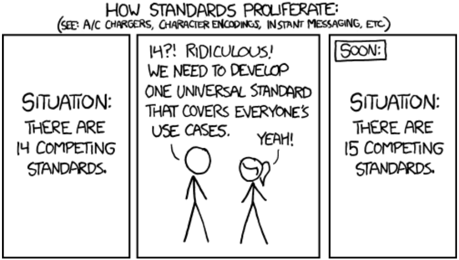
Sebastien Naud, intern at TSP, is currently working on DNS and Nail.

- 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



Source : `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`

```
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`

```
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)
}
```

- 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

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 $L^\star$

$L^\star$ 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 reasonnable 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

- de Ruiters and Poll, – *Protocol State Fuzzing of TLS Implementations* (USENIX Security 2015)
- `https://www.usenix.org/node/190893`

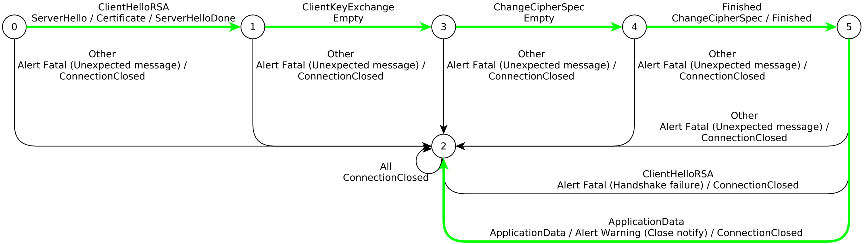
H2

- Georges Bossert – *Comparaisons et attaques sur le protocole HTTP2* (SSTIC 2016)
- `https://www.sstic.org/2016/presentation/comparaisons_attaques_http2/`

SSH

- Fiterau-Brostean et al. – *Model Learning and Model Checking of SSH Implementations* (SPIN'17)
- `https://www.cs.ru.nl/E.Poll/papers/learning_ssh.pdf`

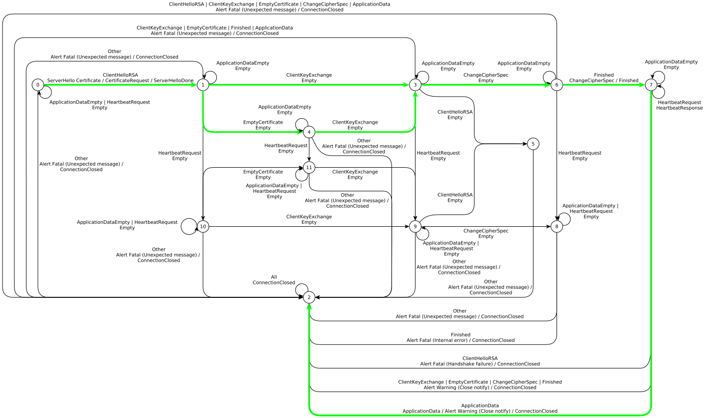
Example of a discovered flaw (1/2)



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 $L^\star$

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 $L^\star$

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

$L^\star$

- better understand `pylstar`
- or implement a new version of $L^\star$?
- improve the performance with a grey-box approach

Next steps (2/3)

Use the message parsers to work on several ecosystems (network scans, implementation tests)

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

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

X.509	rather complete description
SSL/TLS	most TLS < 1.3 messages rudimentary TLS 1.0 implementation
Kerberos	PKINIT messages
BGP/MRT	tool to extract the prefixes announced
DNS	tutorial + `picodig`
NTP	several messages

TAR	tutorial
PNG	tutorial
OpenPGP	packet structure
DVI	simple dissection