Parsing & Understanding in a Messy World

Mike Dodds - January 2024

galois

Context: Galois / me

Galois: A contract research shop. Paid research by-the-hour

- Security / reliability technologies (PL, formal methods, static analysis)
- *Clients:* DARPA, US Gov, some commercial

Me: verified cryptography, formal methods at scale, parser security

- *2004 → 2017:* York / Cambridge / York – PhD, postdoc, junior professor
- *2017 → now:* Galois principal scientist (~ full professor)

Context: DARPA SafeDocs project

SafeDocs: Galois + other teams try to make parsing better

Galois built two tools:

- Format Analysis Workbench (FAW), a tool for understanding existing formats
- Daedalus, a language for developing safer parsers

We have:

- built a high-assurance parser which covers most real-world PDFs
- analyzed millions of real-world PDF documents
- fixed multiple issues in the PDF standard

This talk:

1. Parsing matters a lot and is very hard
2. *Eg:* PDF, an interesting and horrible format
3. Two core problems in safer parsing
4. Some progress: FAW & Daedalus

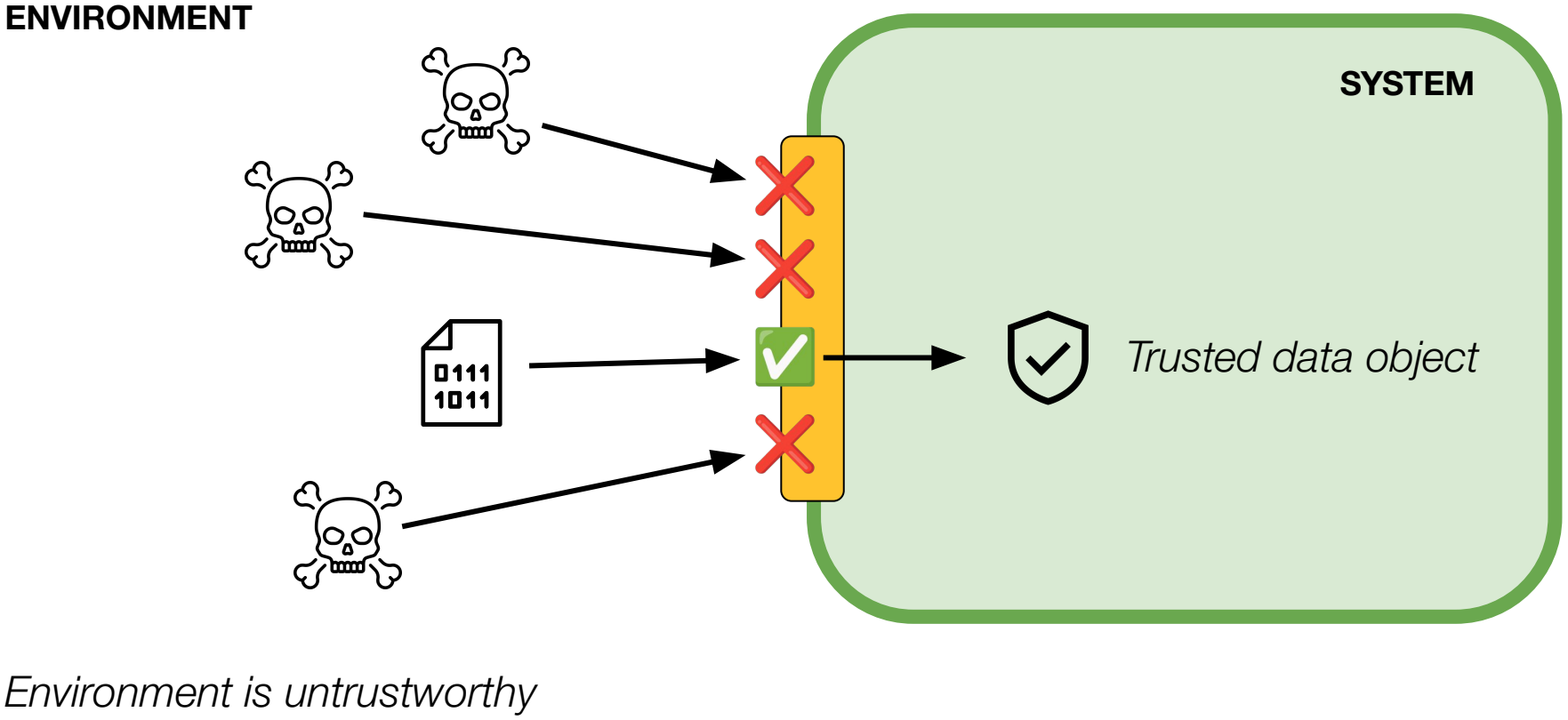
1. Parsing matters a lot and is very hard

Parsers are the immune system

A system has an outside *(low trust)* and an inside *(high trust)*

Systems interact with the world

Parsers convert low-trust data to high-trust data

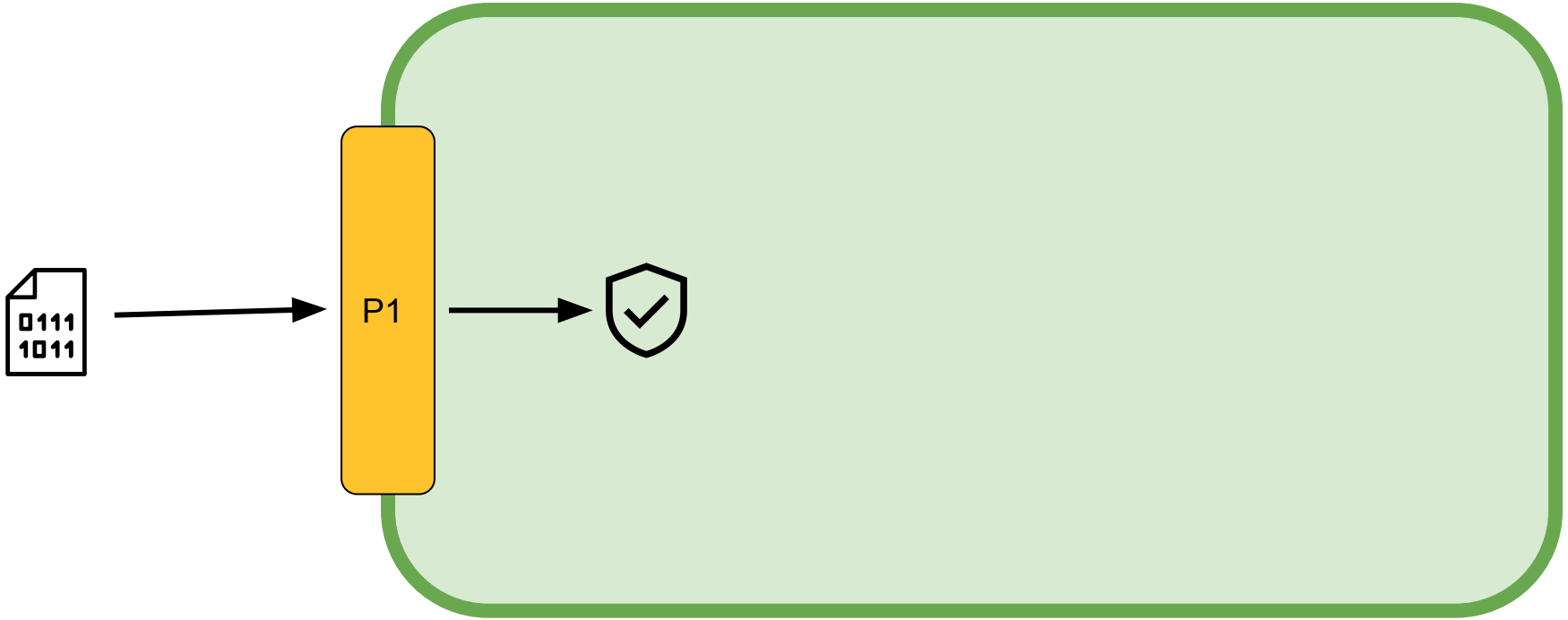


Environment is untrustworthy

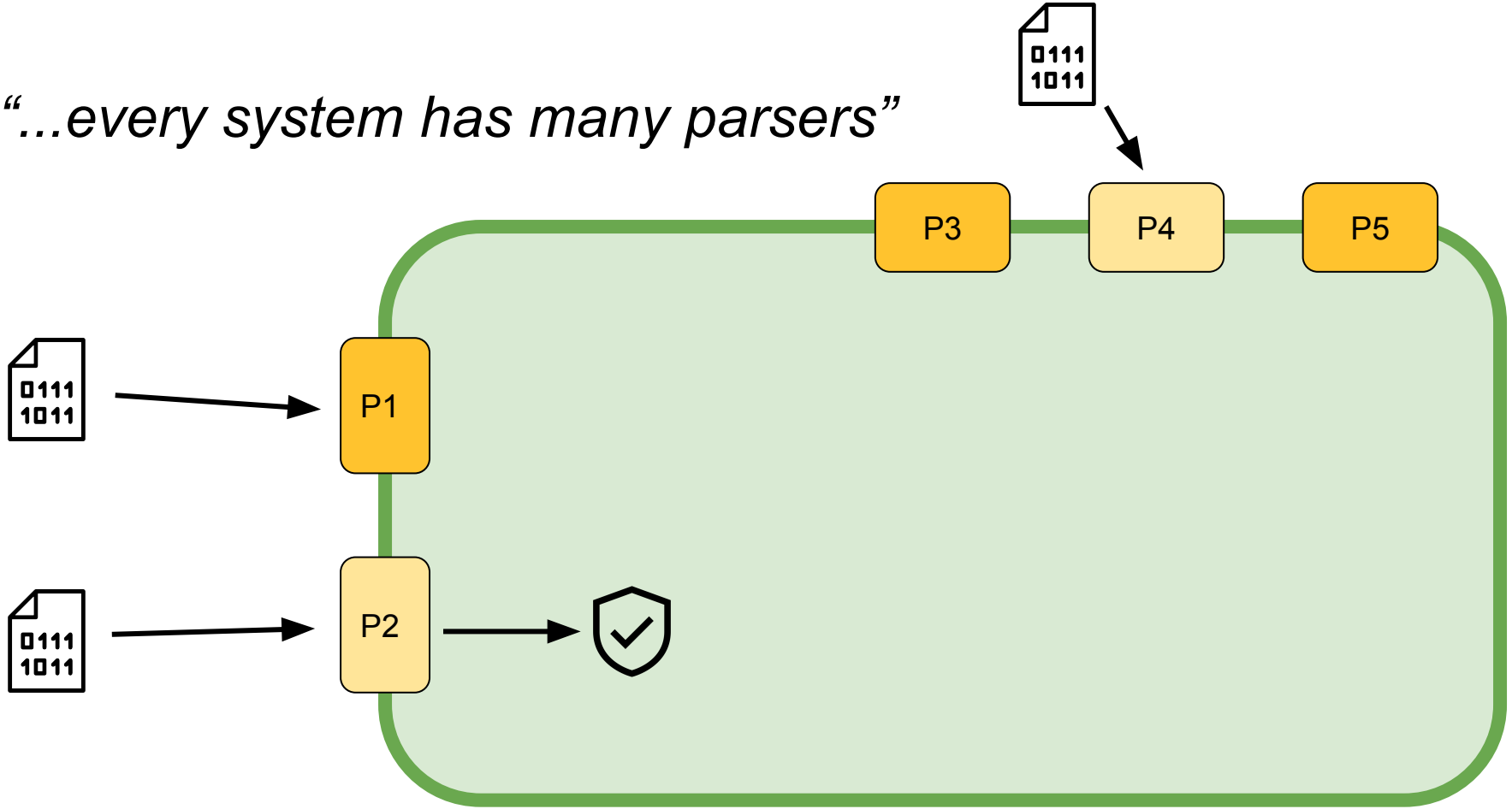
https://www.flaticon.com/free-icon/skull_2099721
https://www.flaticon.com/free-icon/shield_1489589
https://www.flaticon.com/free-icon/binary-code_274412

Everyone writes parsers

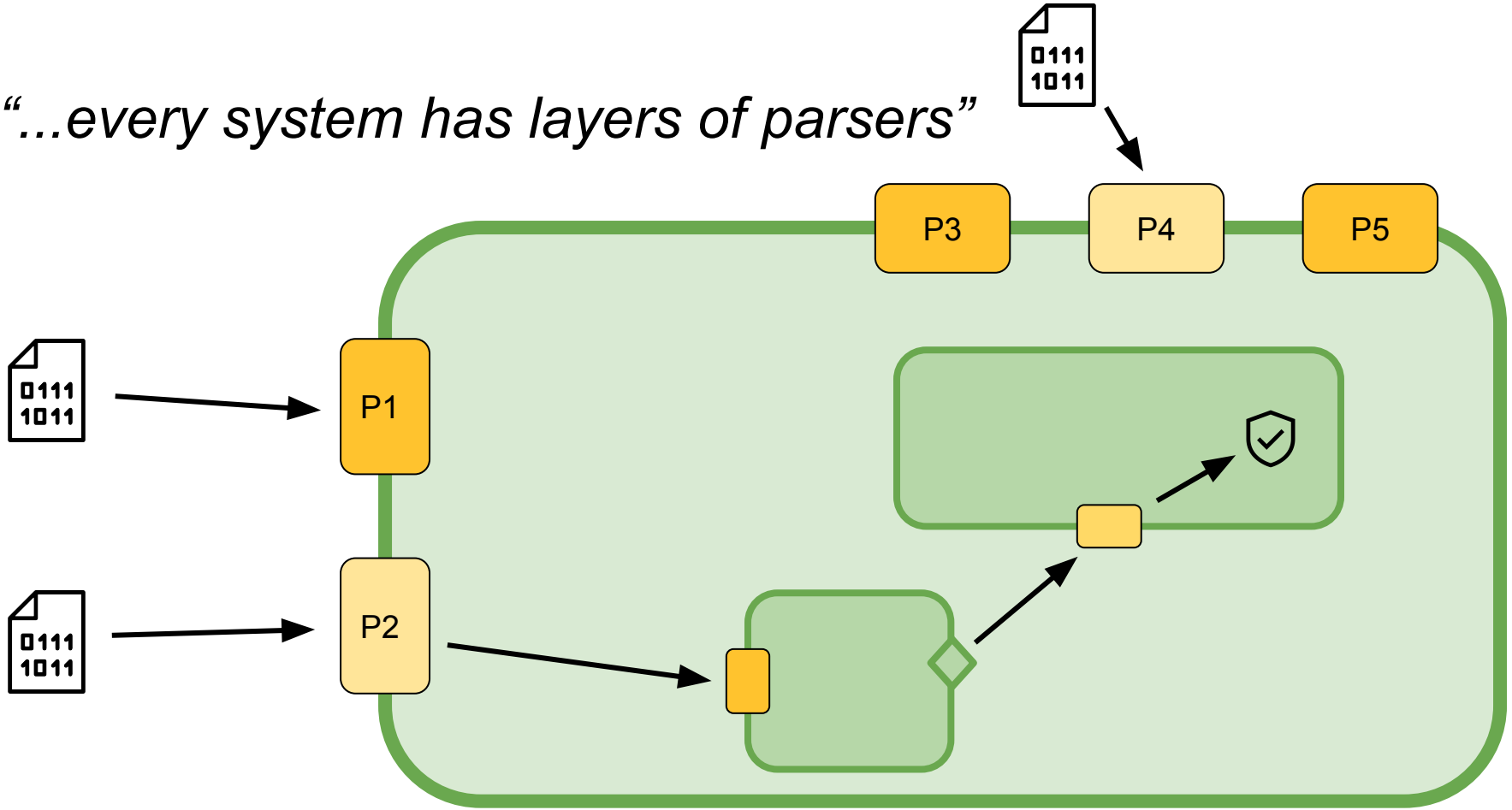
Me (naive): *“Every system has a parser”*



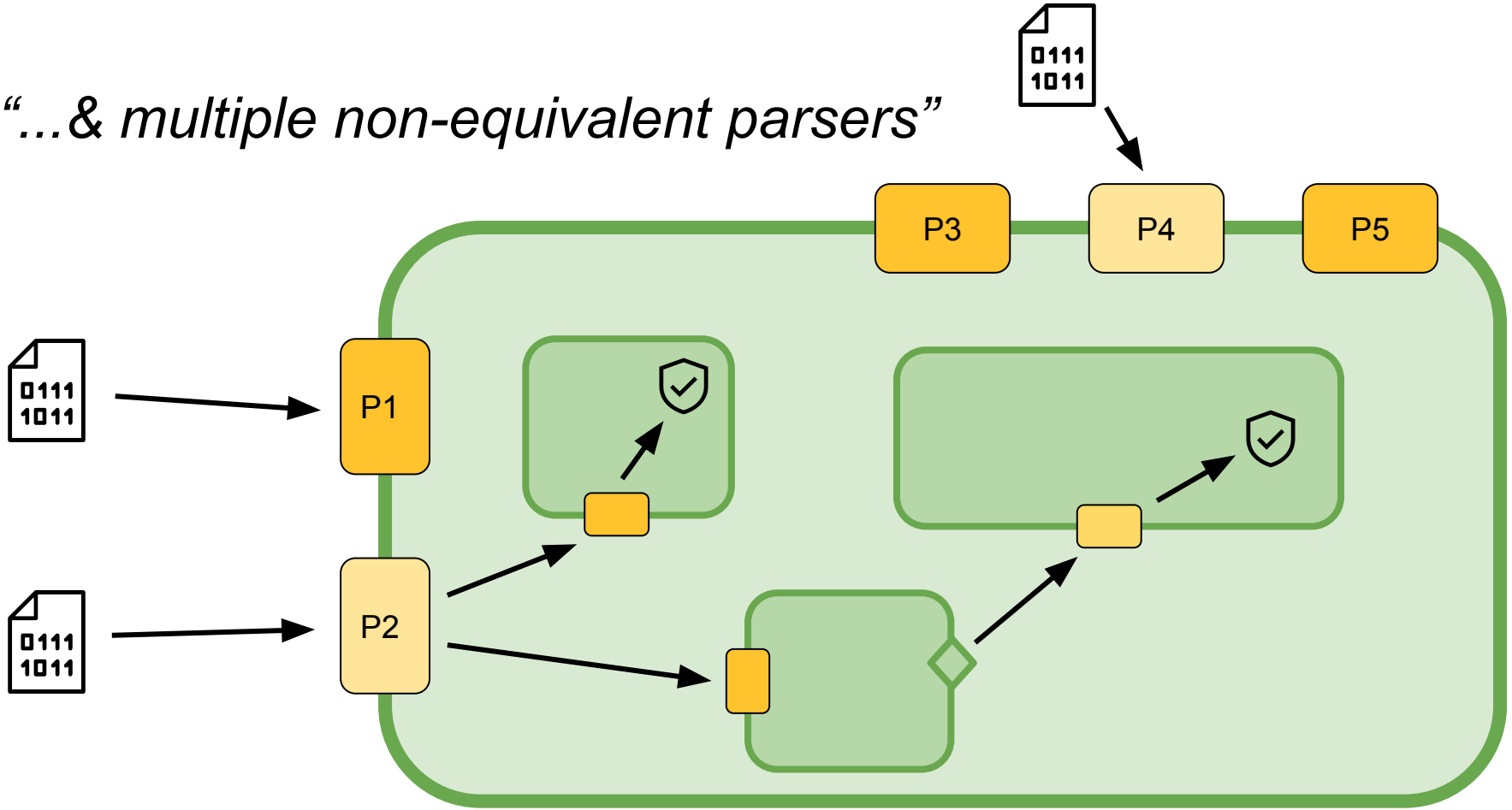
"...every system has many parsers"



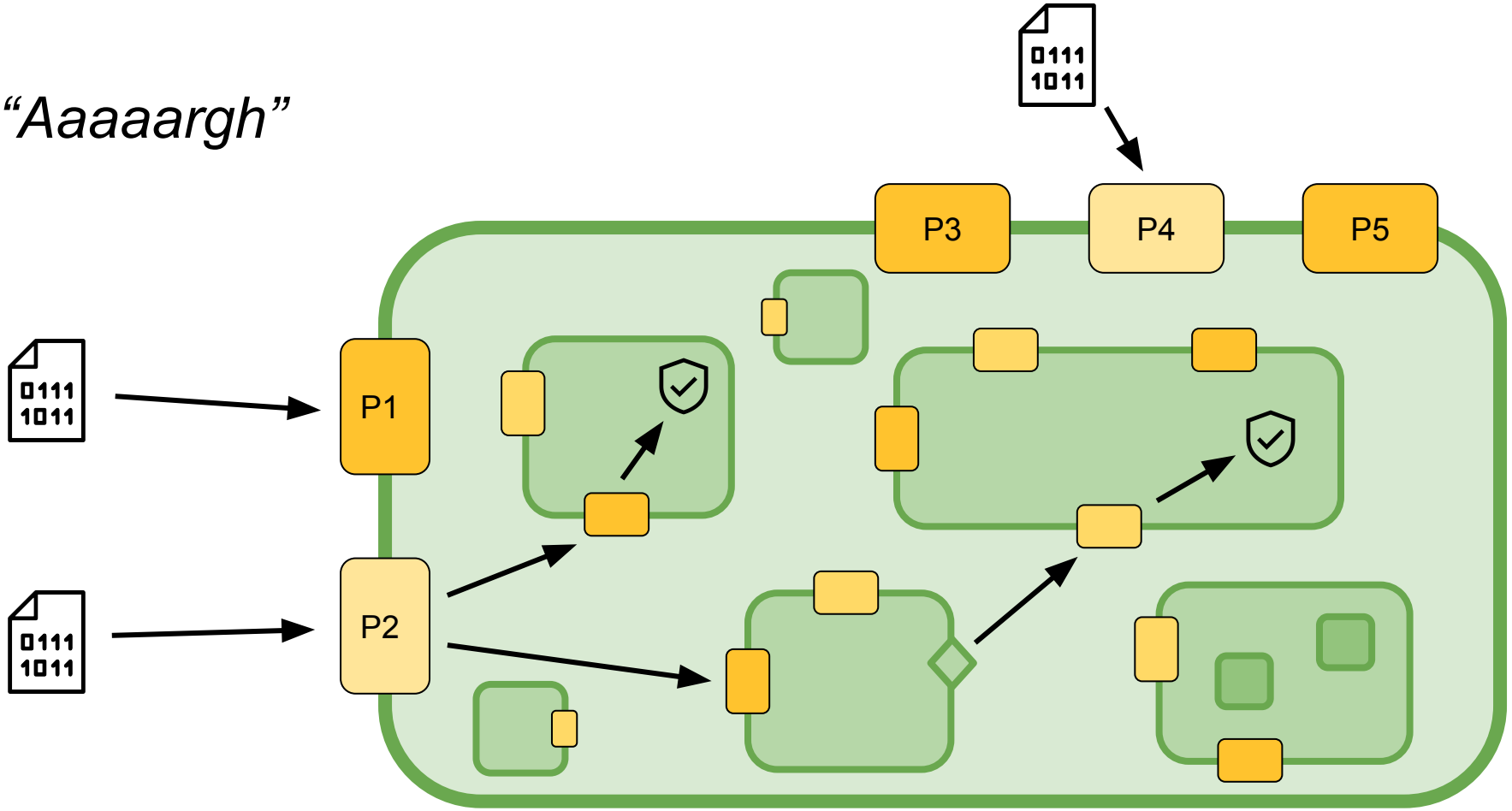
“...every system has layers of parsers”



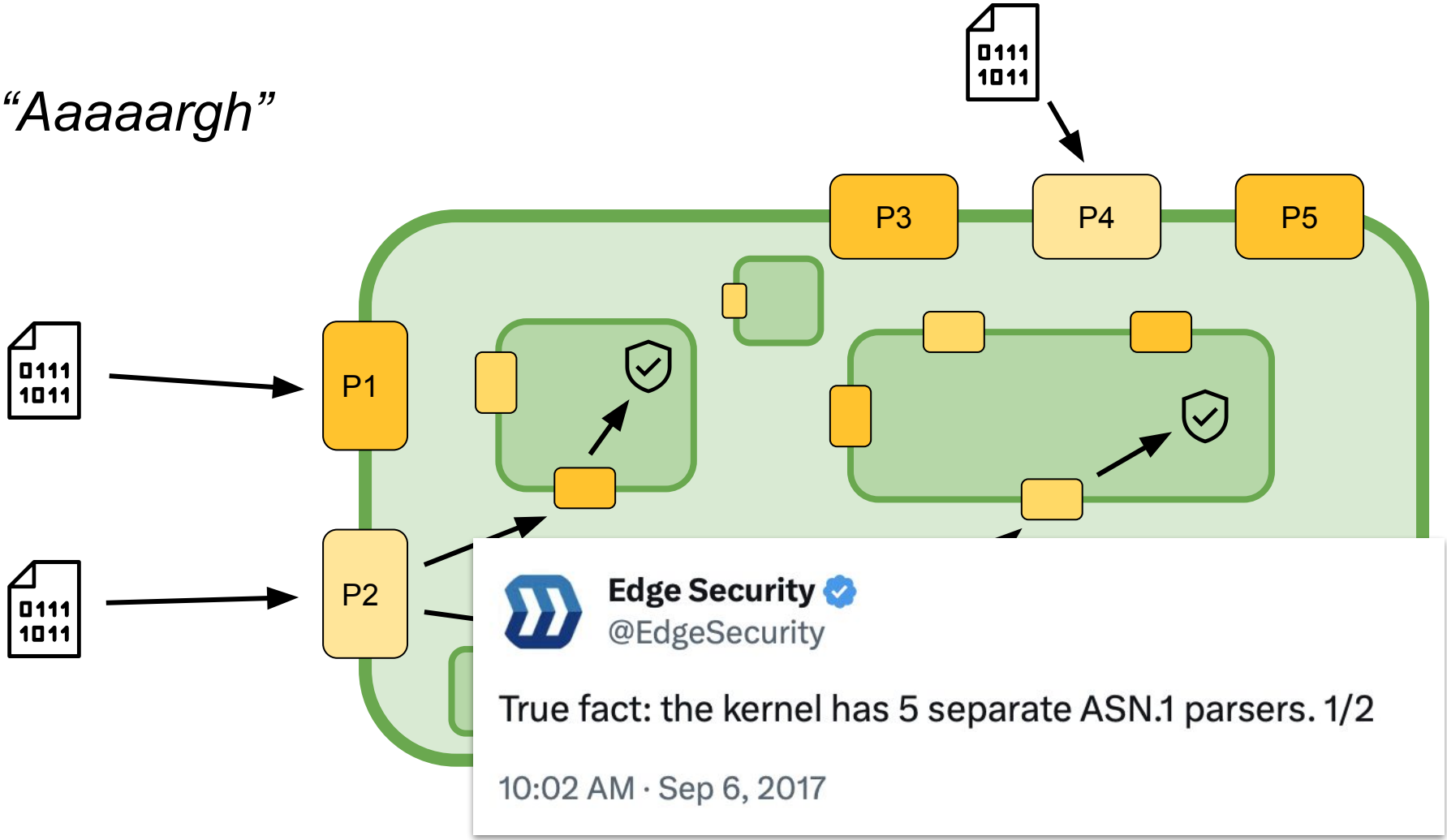
"...& multiple non-equivalent parsers"



“Aaaaargh”



“Aaaaargh”


0111
1011
P3
P4
P5
P1
P2
Edge Security
@EdgeSecurity
True fact: the kernel has 5 separate ASN.1 parsers. 1/2
10:02 AM · Sep 6, 2017

Why are there so many parsers?

Parsers are:

- performance critical
- intermingled with computation
- perform different tasks (security filter, data parsing, constructing values)
- written in languages don’t provide clean abstractions

Also: systems are built over time, and parsers tend to grow capabilities

Parsers fail in interesting ways

1: Parsers crash

This is quite bad

Usually this means a *memory safety violation*

Potentially, this allows an adversary to write into memory

Desired property: *absence of undefined behavior*

No specification required. A crash is a crash.

2: Parsers construct semantic values incorrectly

This is bad, obviously

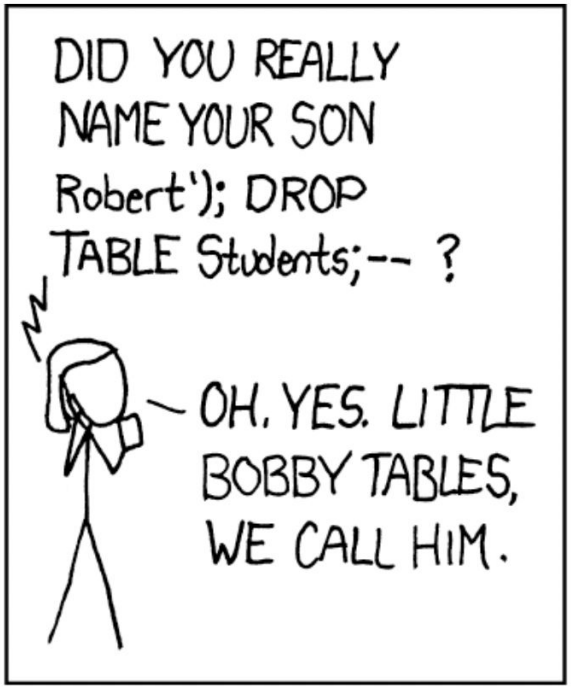
Eg. data sanitization failures - see →

But also, meaning is reconstructed wrongly

Desired property: *the parser behaves as intended*

We need to know what is intended

(...even for non-conformant inputs)



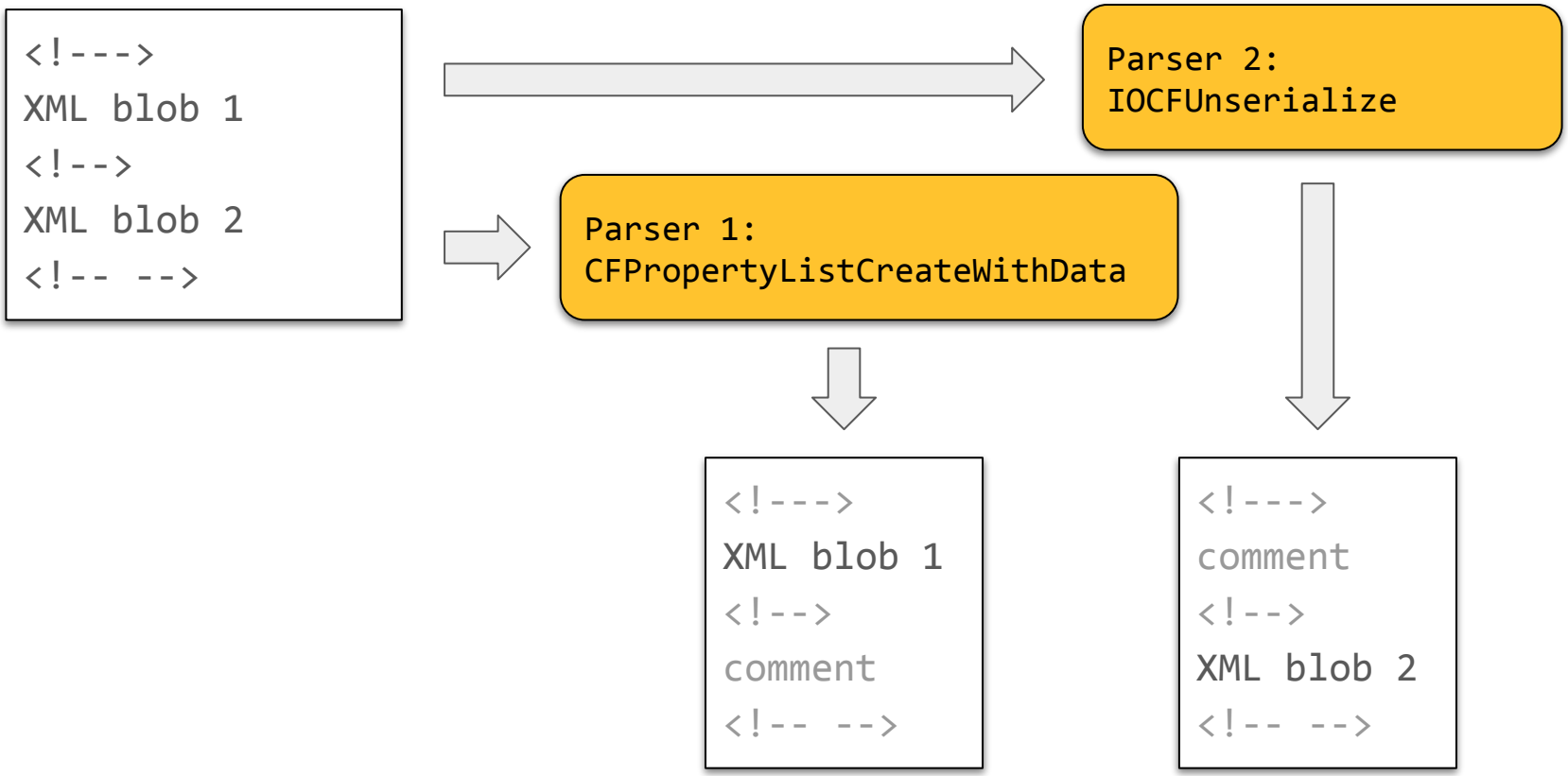
https://xkcd.com/327/

3: Parsers disagree

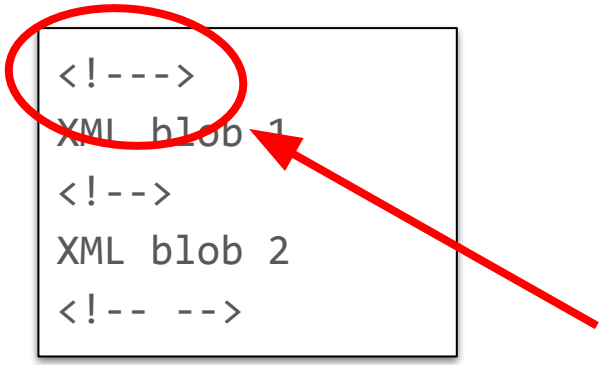
Er, this might be bad?

Actually, let me give an example…

Aside: 'psychic paper' https://blog.siguza.net/psychicpaper/



Aside: 'psychic paper' https://blog.siguza.net/psychicpaper/



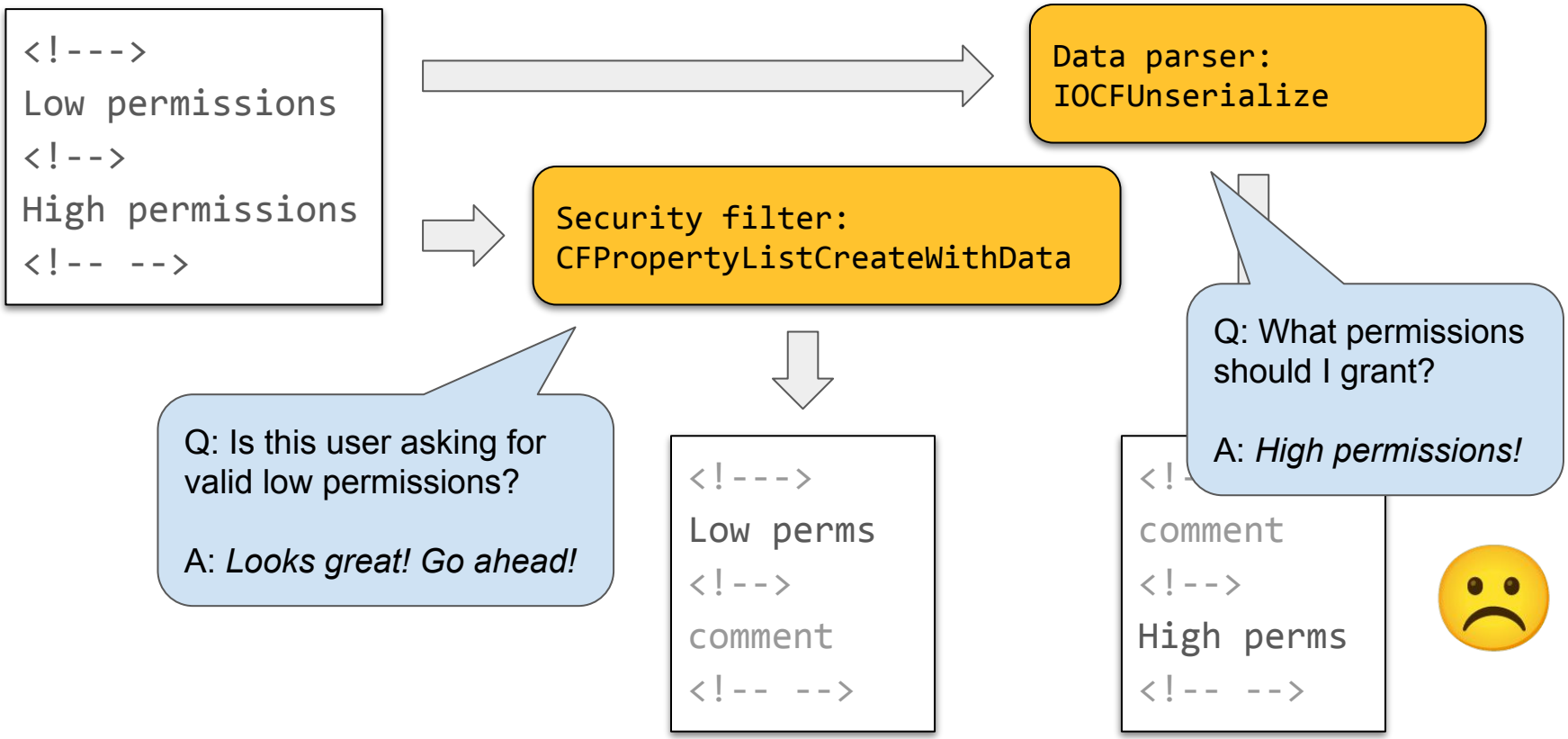
The culprit:

Not valid XML. It could be interpreted as:

- "Start comment"
- "Start *and end* comment"

⇒ Both parsers are 'correct'!

Aside: 'psychic paper' https://blog.siguza.net/psychicpaper/



3: Parsers disagree

So this is actually bad, and hard to detect

Examples:

- Sneak past security parsers
- PDFs that parse differently when viewed and printed

Desired property: *parsers agree with each other*

This is a meta-property between parsers

Writing correct parsers is very hard

Ground truth does not exist

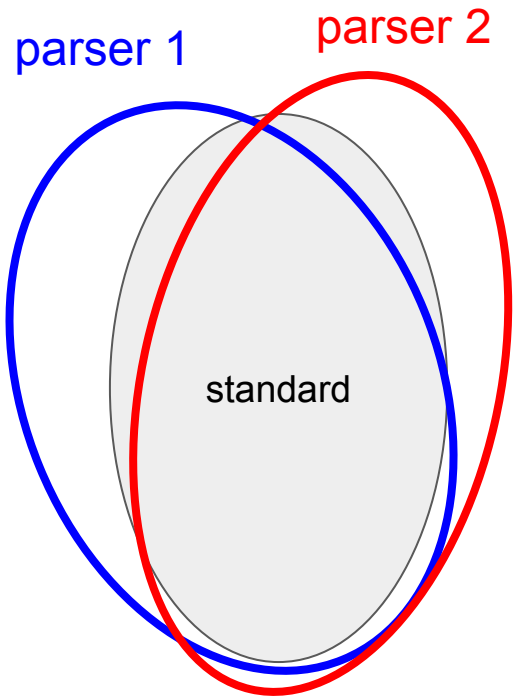
We can assume:

- One or more existing parsers
- Some documentation, and maybe a standard
- A set of examples of the format

We can’t assume:

- Agreement between existing parsers
- Specifications matching *de facto* behavior

Parsers are incentivized to parse non-conformant inputs



Common parser languages are difficult to audit

Typically, parsers are written in C++ & similar:

- Hard to even establish absence of undefined behavior
- Hard to extract parser behavior / reason about parsers
- Hard to specify parsers at the high level
- Hard to audit behavior for humans

Fuzzing is good but limited

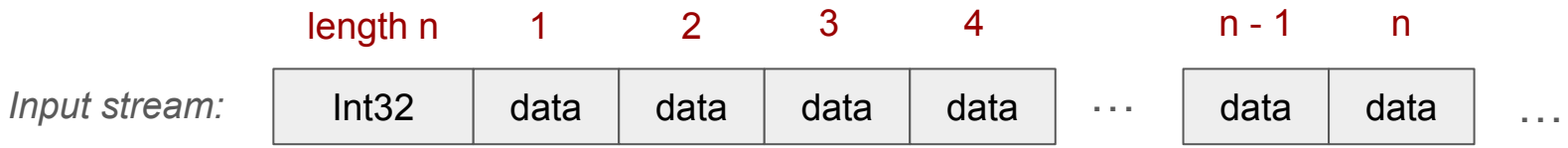
- Generate lots of random inputs
- Guided search for crashes

Fuzzing is de-facto the way that parsers are secured

But: only finds *undefined behavior* parser flaws

Hold on didn’t we solve parsing in, like… 1959?

Damn you, data dependency



We have to:

- Parse length, and compute n
- Read n more data chunks

In general, parsing has to perform *arbitrary computations*

I'll come back to this later…

“Shotgun parsing” *(Brattus et al)*

A common parser structure for dealing with data dependency:

- Read some data
- Call some arbitrary handler function (e.g written in C++)
- Return a value and keep going

Unsafe, hard to maintain, non-auditable

2. PDF, an interesting and horrible format

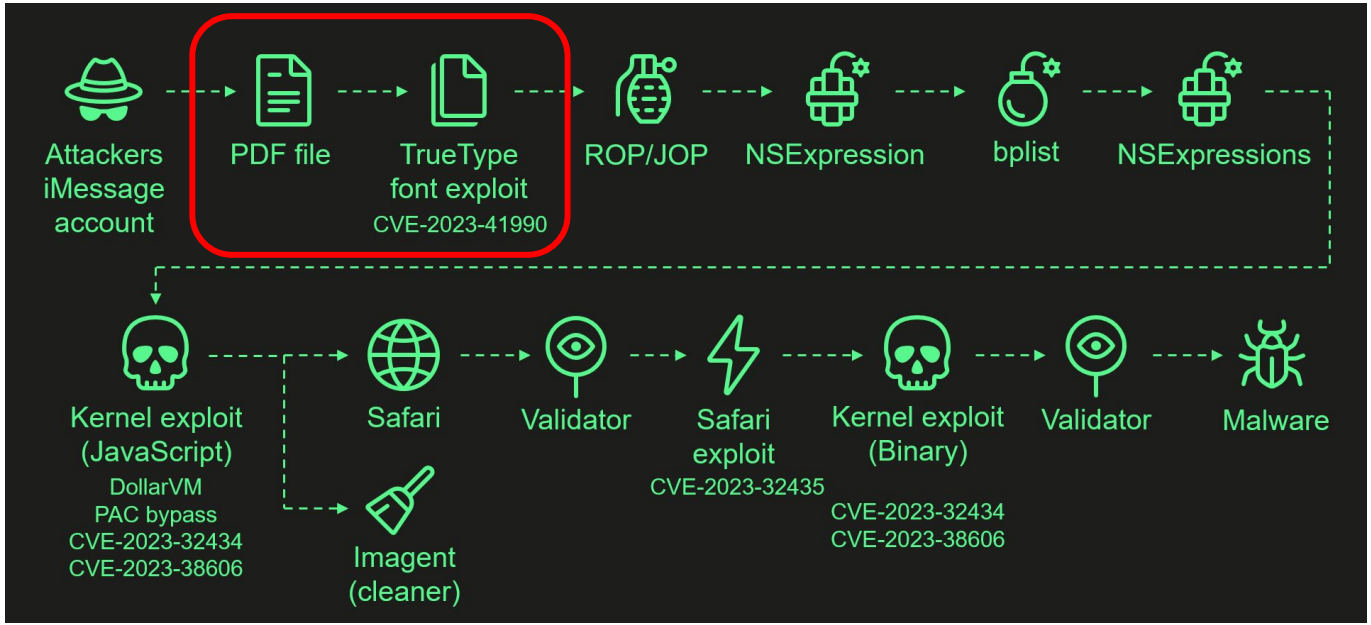
PDF is important and interesting

- Billions of users
- *De facto* message format for many human processes
- Huge attack surface & many vulnerabilities
- Contains embedded formats: images, fonts, JavaScript, video *(...yes, really)*

- Has a somewhat agreed core standard
- Many real implementations - some good, some v bad
- Huge dataset of examples in the wild

PDF is an attack vector

Eg. Operation Triangulation (December 2023) https://securelist.com/trng-2023/



Operation Triangulation exploit chain

SafeDocs built a huge dataset of PDFs

https://pdfa.org/new-large-scale-pdf-corpus-now-publicly-available/

About 8m extant (real-world) documents

A large proportion of these PDFs don’t conform to the standard

PDF is weird
and hard to parse

XRef table

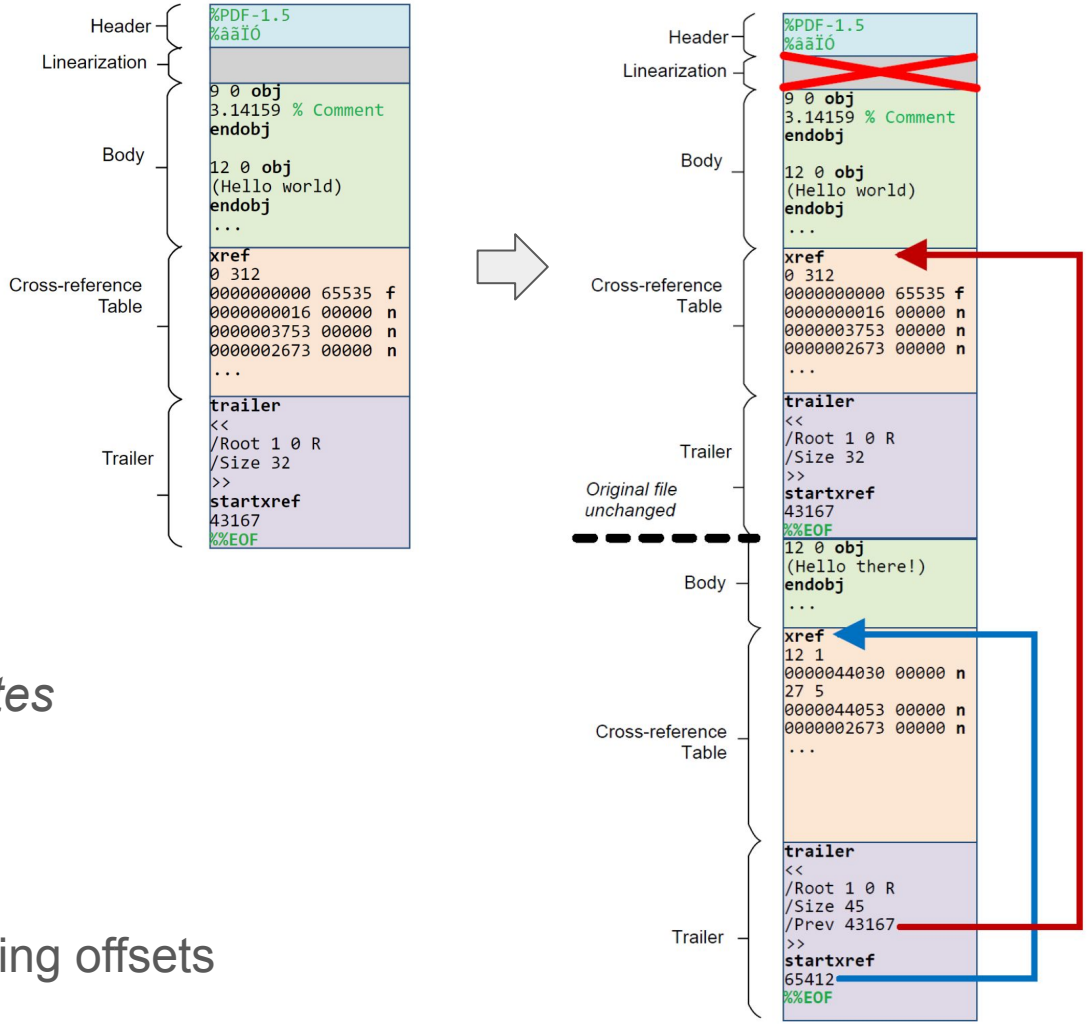
PDF structure:

- A set of objects - text, values, pages, fonts…
- A cross-reference table of object locations

XRef supports *incremental updates*

Surprising results:

- Parsing is highly non-linear
- Parsing depends on computing offsets



Object streams

```
9 0 obj
3.14159 % Comment
endobj

12 0 obj
(Hello world)
endobj
...
```

Objects can be contained in other objects

- Object sizes can be contained in other objects
- Objects can be compressed or encrypted

Surprising results:

- Parsing an object may require accessing multiple other objects
- Parsing may require decrypting / decompressing other object

Themes in parsing PDF:

- Pervasive data-dependency
- Pervasive computation
- Non-local parsing
- Many embedded formats

Hypothesis:
most mature formats
are super weird

3. Two core problems in safer parsing

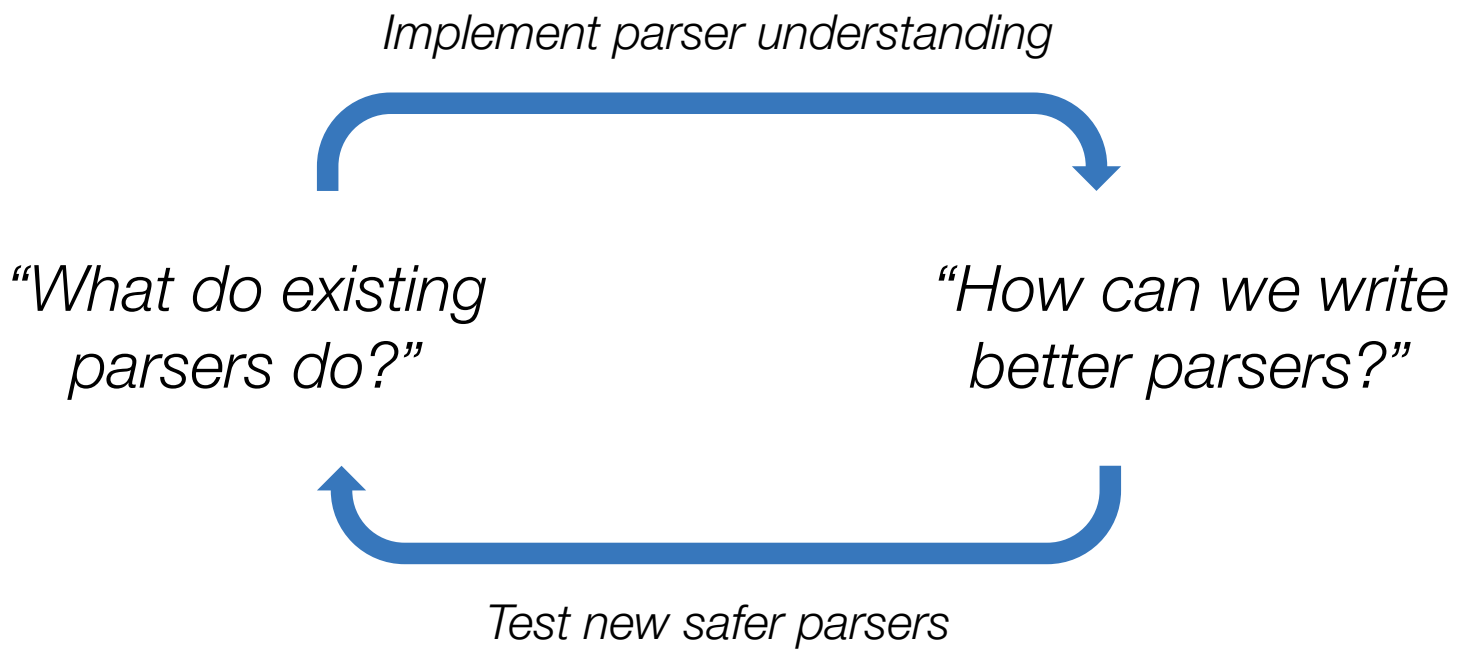
“What do existing parsers do?”

“How can we write better parsers?”

Implement parser understanding

“What do existing parsers do?”

“How can we write better parsers?”


Implement parser understanding
“What do existing parsers do?”
“How can we write better parsers?”
Test new safer parsers

4. Some progress (Daedalus and FAW)

“What do existing parsers do?”

“How can we write better parsers?”

Format Analysis Workbench

An *investigation engine* for understanding parsers and formats

FAW can:

- Run parsers at scale
- Analyze results
- Test hypotheses
- Generate understanding

Daedalus

A *format description language* for generating safe and correct parsers

Daedalus can:

- Define human-readable format definitions
- Prevent crashes
- Synthesize parsers

Format Analysis Workbench

An *investigation engine* for understanding parsers and formats

FAW can:

- Run parsers at scale
- Analyze results
- Test hypotheses
- Generate understanding

Daedalus

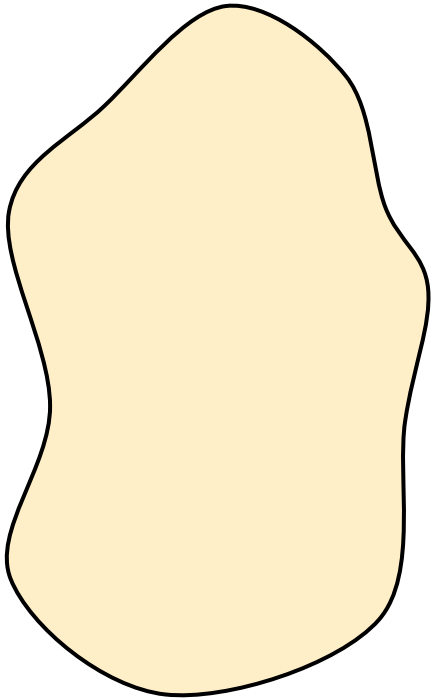
A *format description language* for generating safe and correct parsers

Daedalus can:

- Define human-readable format definitions
- Prevent crashes
- Synthesize parsers

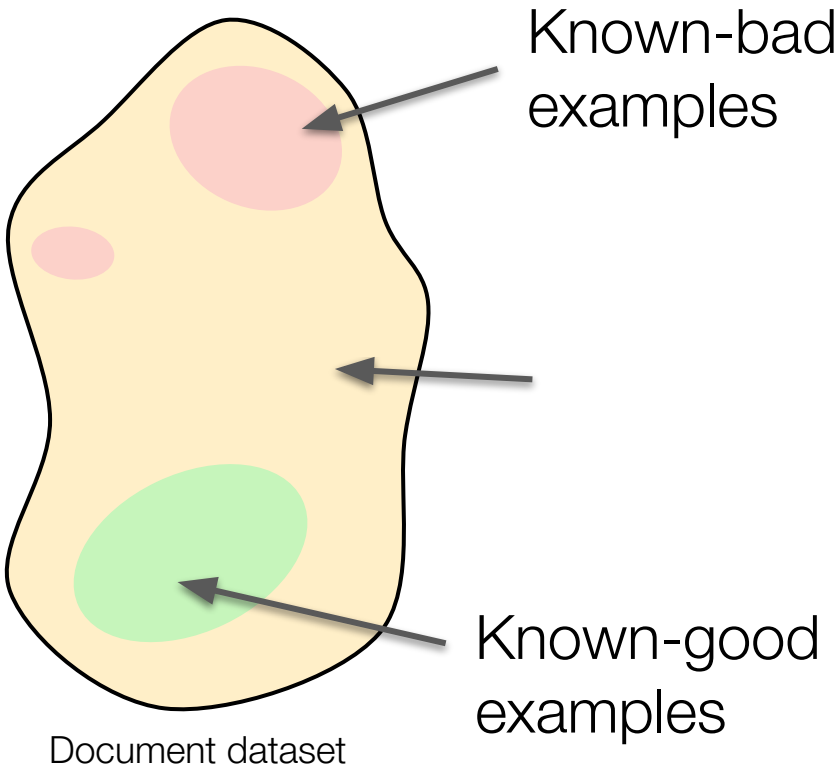
Format Analysis Workbench (FAW)

“What do existing parsers do?”



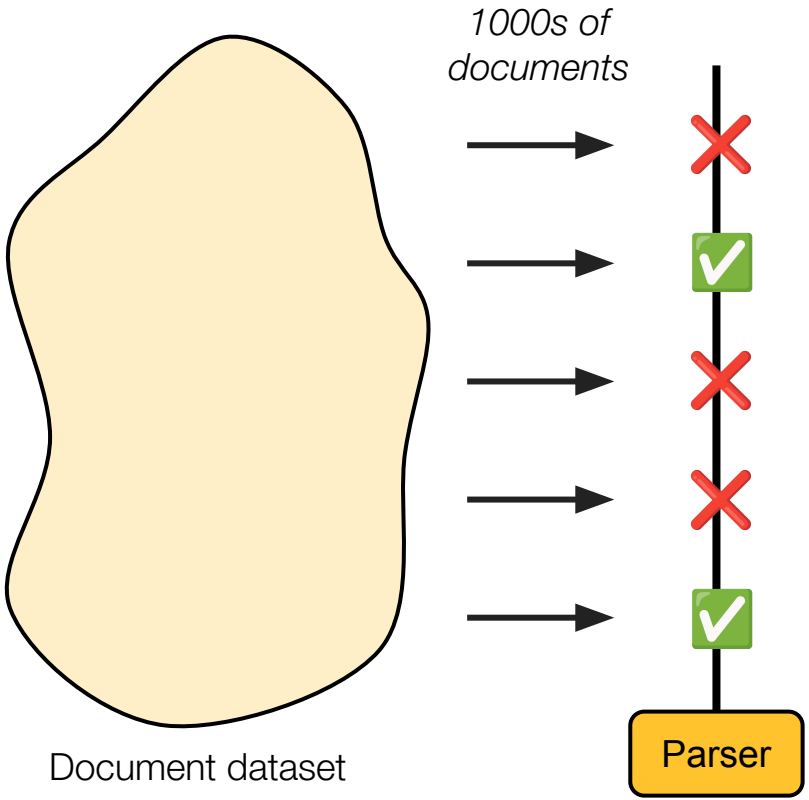
Document dataset

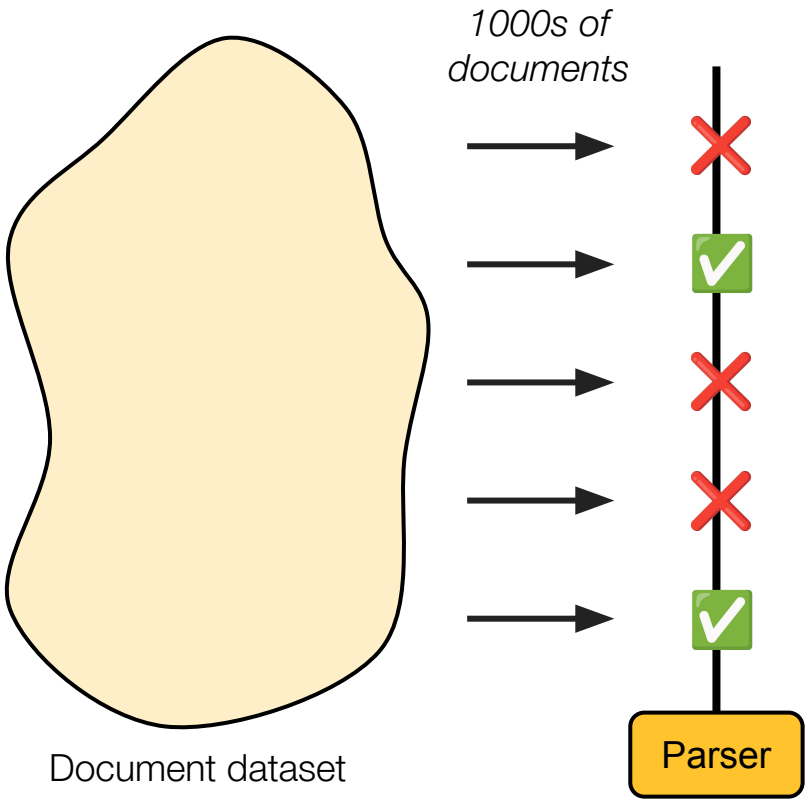
Document dataset



Eg, PDF dataset:

- 1M+ files
- Some known-good and known-bad examples, but mostly unknown


1000s of documents
Document dataset
Parser

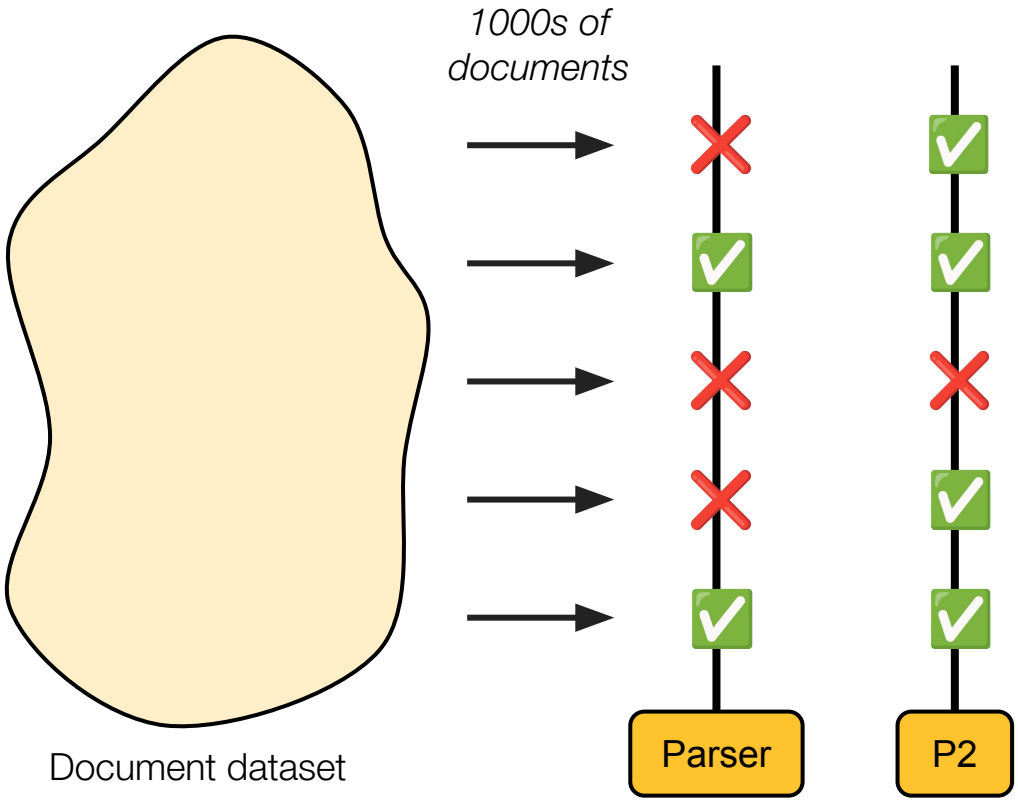


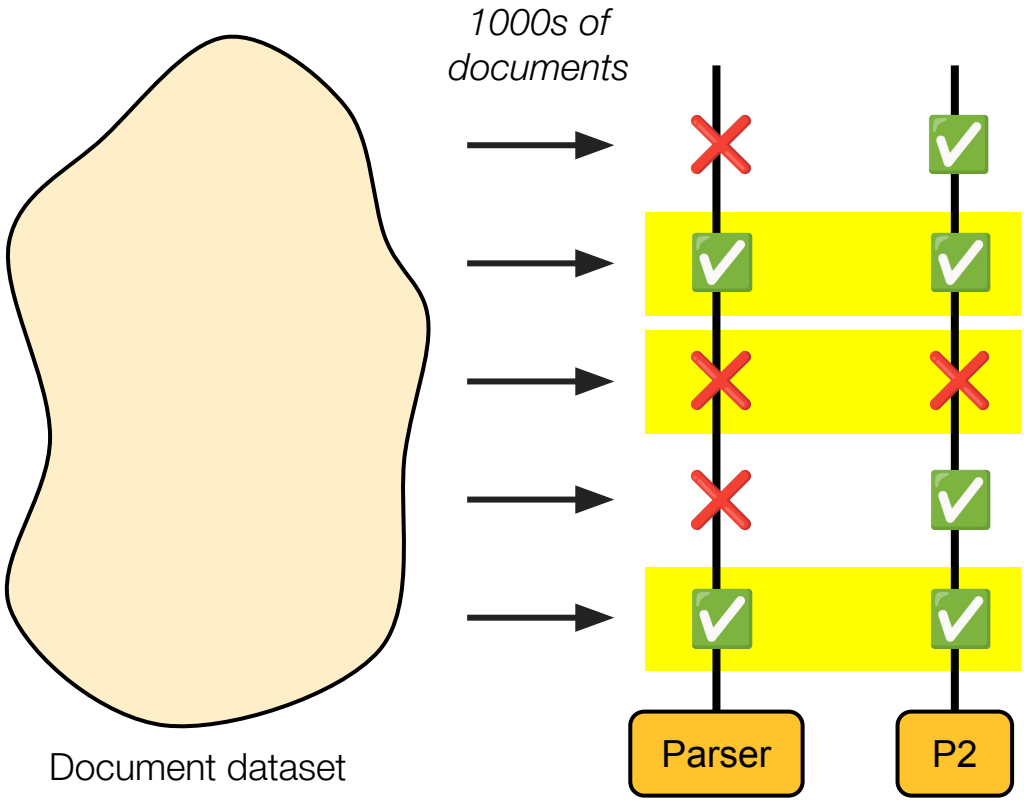
Results of parsing:

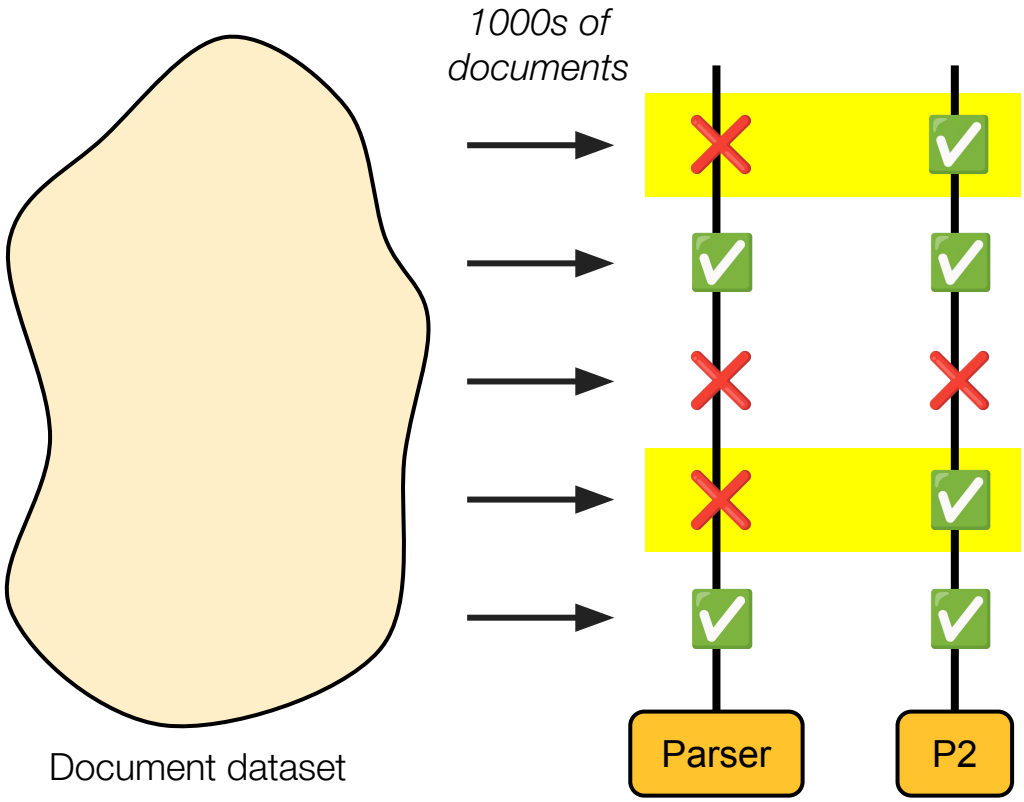
- Valid ✅ or invalid ❌
- Parser return codes

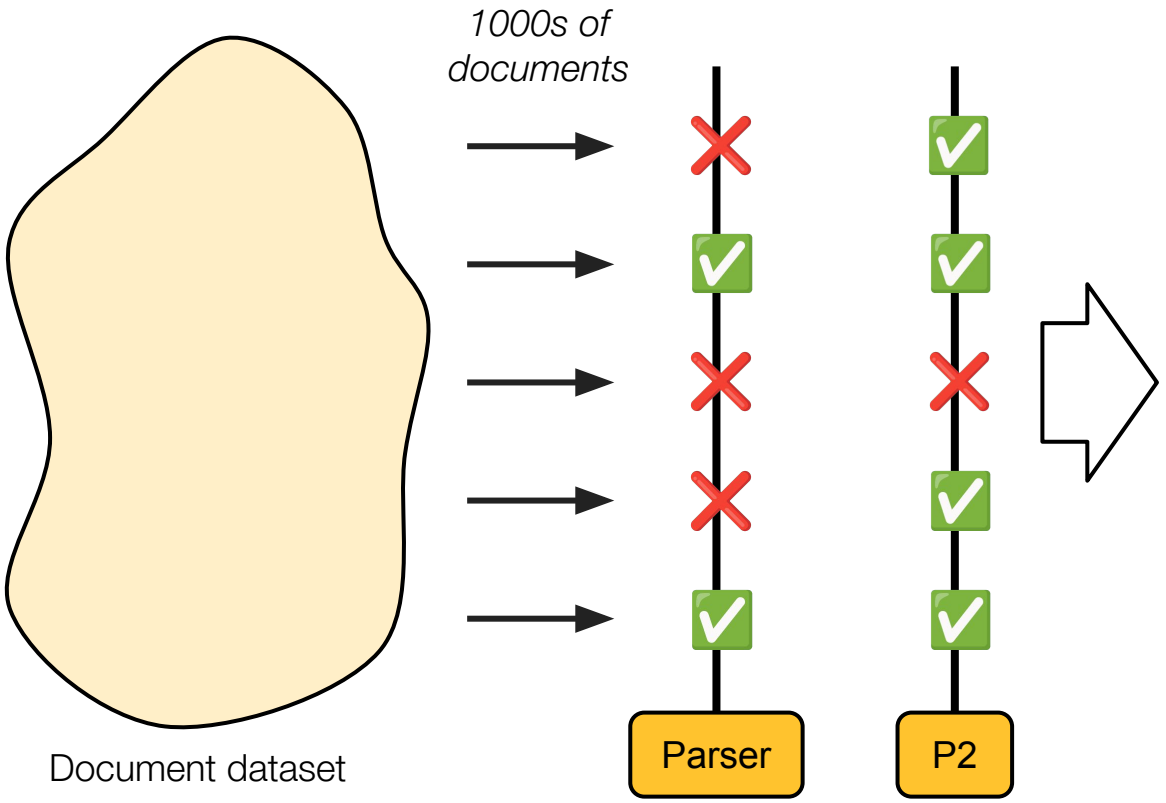
But also:

- Plug-in analysis results
- Any tool that can apply to a parser!


1000s of documents
Document dataset
Parser
P2


1000s of documents
Document dataset
Parser
P2


1000s of documents
Document dataset
Parser
P2



Parser \\ P2	rejected	valid
rejected	X %	A %
valid	B %	Y %

The FAW is a format science lab

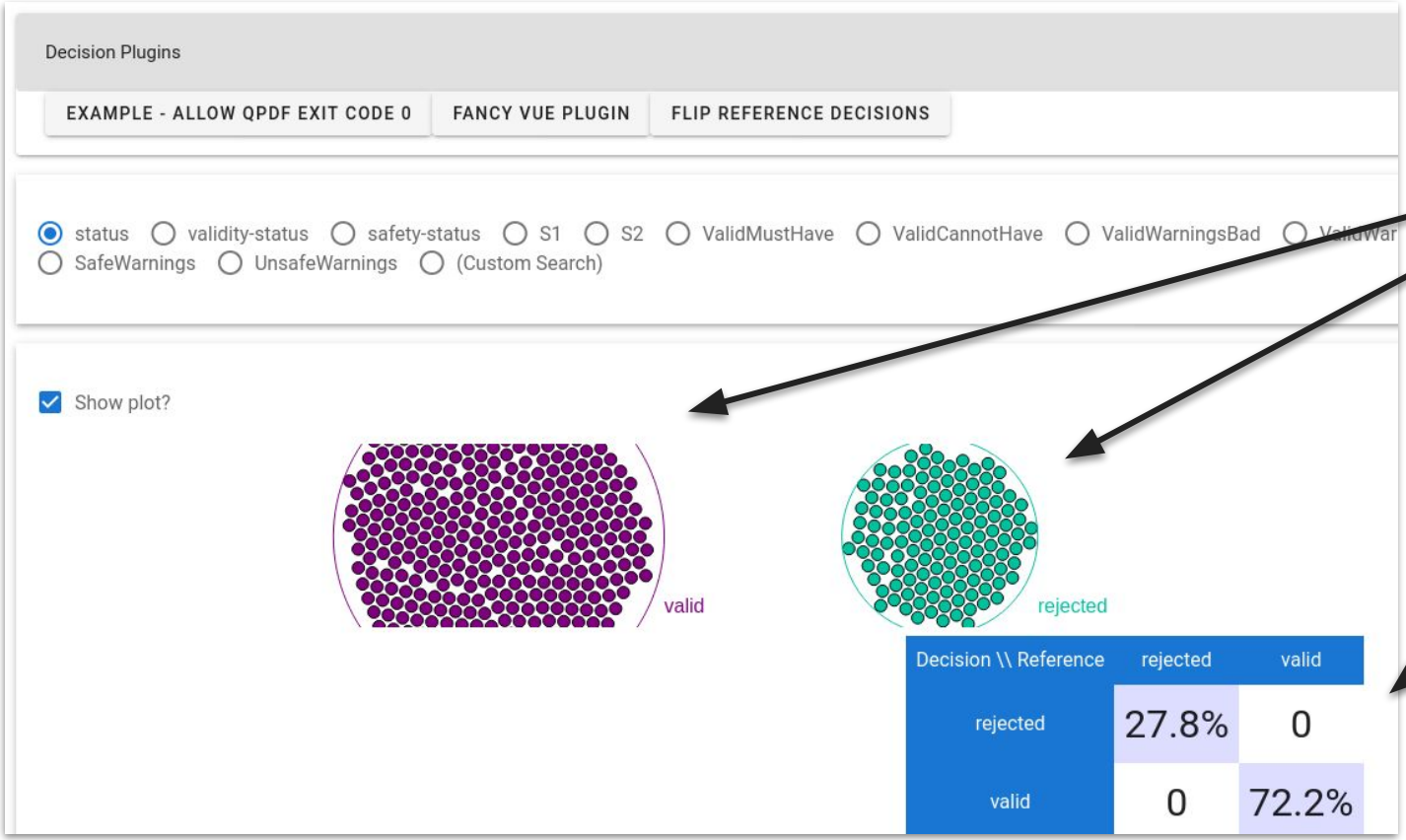
Inputs:

- Format examples (e.g., PDFs)
- Parsers or programs that ingest those examples

Use cases:

- Identify potentially unsafe inputs
- Identify causes of false alarms at scale
- Understand patterns of input and how they affect individual programs at a deep level

FAW interface



Outcomes of parsing by result type

Breakdown of results

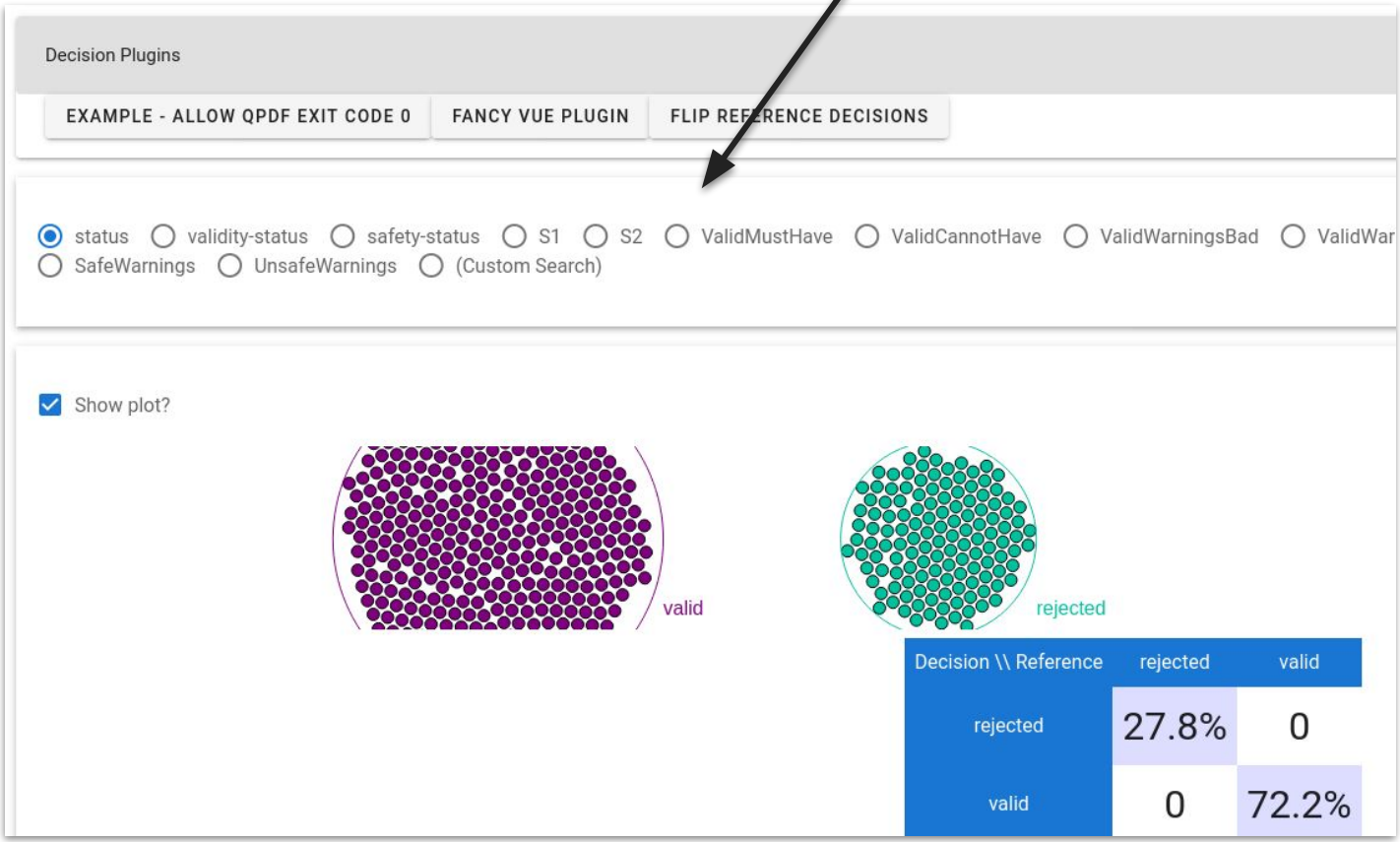
Categorization is controlled by the user

```
outputs:
  # Standard output status -- If a PDF passes filter S1, it will be "valid",
  # otherwise "rejected".
  status:
    "valid" is !(RejectedBad | RejectedAmbiguousBad | ValidWarningsXrefRebuild & XrefIsAmbiguous)
    "rejected" else

  # Mark anything that is linearized as "rejected-unsafe", and otherwise
  # mark it as "valid".
  validity-status:
    "rejected" is RejectedBad
    "rejected-ambiguous" is RejectedAmbiguousBad | (ValidWarningsXrefRebuild & XrefIsAmbiguous)
    "valid-warnings" is ValidWarningsBad | !ValidMustHave | ValidCannotHave | ValidWarningsXrefRebuild
    "valid" else
    "rejected-unsafe"

  safety-status:
    "unsafe"
    "unsafe-warnings" is UnsafeWarnings
    "safe-warnings" is SafeWarnings
    "safe" else
```

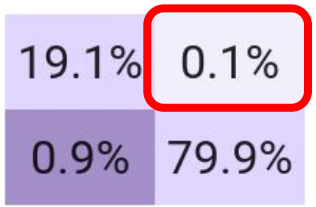
Output status is determined by a regexp-based *alarm language*

FAW interface

Output status controls



The FAW assists in format detective work



FILTER IN FAW

Show error messages causing reject:

parser-mupdf-mutool-draw-ppm_error: cannot draw '<inputFile>': 1 / 0
parser-mupdf-mutool-draw-ppm_page <inputFile> error: cannot fwrite: No space left on device: 1 / 0
parser-xpdf-pdftoppm_I/O Error: Couldn't create temporary font file: 1 / 1
parser-mupdf-mutool-clean_error: cannot fwrite: No space left on device: 3 / 3

Manually inspect the failing files

Identify a dataset for further investigation

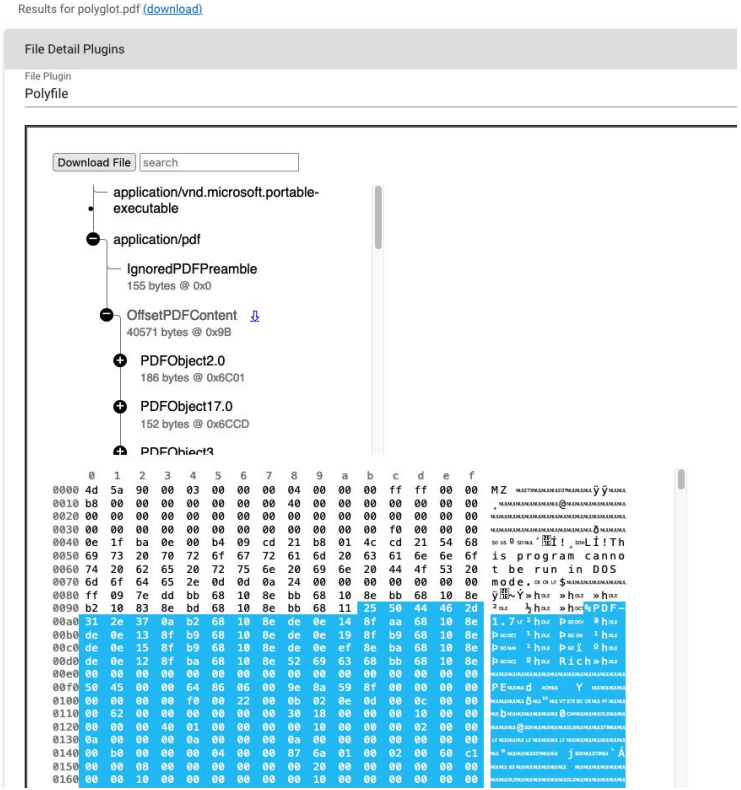
Run further analyses and discover correlations

Interrogation example: PolyFile

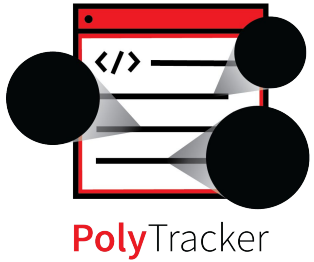
A utility by *Trail of Bits* for examining the structure of files and detecting their file type

- Hex viewer for examining the file in detail which shows how various parts of the binary are interpreted
- Map file data back to AST nodes generated by the parser
- Plugs into FAW

https://github.com/trailofbits/polyfile



Interrogation example: PolyTracker



https://github.com/trailofbits/polytracker

PolyTracker: a generic taint tracking tool

- Binary instrumentation to track data processing
- Identify which parser functions touch which parts of the file

PolyFile + PolyTracker:

- Map a file's meaning (PolyFile)
- Map how it is used in the binary (PolyTracker)

“What do existing parsers do?”

Implement parser understanding

“What do existing parsers do?”

“How can we write better parsers?”

“How can we write better parsers?”

Daedalus

Daedalus: a language for writing formats

Aim: *close the gap from formats to parsers*

- Powerful enough to represent eg. PDF
- Amenable to human reading and static analysis
- Type-safe, crashes can’t happen
- Turing-complete, but highly structured
- Amenable to performant compilation into C++

Daedalus is a language and toolchain

Daedalus *(language)*: Data Description Language

Daedalus *(toolchain)*: compilation and execution of Daedalus-lang specifications

- Compile Daedalus to performant C++ code

Daedalus design

Based on functional programming ideas / parser combinators

Includes several highly useful capabilities:

- A generic notion of data dependency. Depend on any datatype
- An encapsulated notion of an input stream. Safe non-linear parsing.
- An FFI interface. Call into helper functions in a controlled way

Example: PPM, a small image format

Specification:

- A magic number identifying the file type (for ASCII PPM, this is *P3*)
- The dimensions of the image (width then height)
- The maximum color value
- A ‘matrix’ of RGB triples for each pixel defined in row-major order

A PPM file

```
P3
4 4
15
 0  0  0    0  0  0    0  0  0   15  0 15
 0  0  0    0 15  7    0  0  0    0  0  0
 0  0  0    0  0  0    0 15  7    0  0  0
15  0 15    0  0  0    0  0  0    0  0  0
```

- The magic number is P3, indicating an ASCII RGB image
- The width and height are both 4
- The maximum color value is 15
- There is a four-by-four grid of triples, one triple per pixel

```
-- PPM format in Daedalus
def Main =
  block
    $$ = PPM

def Token P =
  block
    $$ = P
    Many (1..) WS

def PPM =
  block
    Match "P"
    let version = Token Natural
    version == 3 is true
    width  = Token Natural
    height = Token Natural
    maxVal = Token Natural
    data   = Many height (Many width RGB)

def RGB =
  block
    red   = Token Natural
    green = Token Natural
    blue  = Token Natural

def WS = Match1 (0 | 9 | 12 | 32 | '\n' | '\r')

def Natural =
  block
    let ds = Many (1..) Digit
    ^ for (val = 0; d in ds) (addDigit val d)

def Digit =
  block
    let d = Match1 ('0' .. '9')
    ^ d - '0'

def addDigit val d =  10 * val + (d as uint 64)
```

```
-- PPM format in Daedalus
def Main =
  block
    $$ = PPM

def Token P =
  block
    $$ = P
    Many (1..) WS

def PPM =
  block
    Match "P"
    let version = Token Natural
    version == 3 is true
    width  = Token Natural
    height = Token Natural
    maxVal = Token Natural
    data   = Many height (Many width RGB)
```

Parser declaration in Daedalus

Note that parsers are higher-order - the *Token* parser takes the parameter *P*, itself a parser

```
-- PPM format in Daedalus
def Main =
  block
    $$ = PPM

def Token P =
  block
    $$ = P
    Many (1..) WS

def PPM =
  block
    Match "P"
    let version = Toke
    version == 3 is tr
    width  = Token Na
    height = Token Na
    maxVal = Token Na
    data   = Many height (Many width RGB)
```

Primitive parsing in Daedalus

The parser reads a token “P” off the input stream

If no such token is present, the parser backtracks

Primitive parsing with multiple possible values

The parser *WS* reads one of the possible choices: 0, 9, …

```
def RGB =
  block
    red   = Token Natural
    green = Token Natural
    blue  = Token Natural

def WS = Match1 (0 | 9 | 12 | 32 | '\n' | '\r')

def Natural =
  block
    let ds = Many (1..) Digit
    ^ for (val = 0; d in ds) (addDigit val d)

def Digit =
  block
    let d = Match1 ('0' .. '9')
    ^ d - '0'

def addDigit val d =  10 * val + (d as uint 64)
```

Parser combinators in Daedalus

The *red, green,* and *blue* values are parsed in sequence using the *block* combinator

The return type of the block is a *structure type* with fields *red, green, blue*

```
def RGB =
  block
    red   = Token Natural
    green = Token Natural
    blue  = Token Natural

def WS = Match1 (0 | 9 | 12 | 32 | '\n' | '\r')

def Natural =
  block
    let ds = Many (1..) Digit
    ^ for (val = 0; d in ds) (addDigit val d)

def Digit =
  block
    let d = Match1 ('0' .. '9')
    ^ d - '0'

def addDigit val d =  10 * val + (d as uint 64)
```

Computation in Daedalus

The *Natural* parser reads multiple digits, and then computes the overall value by iterating over the list of digits

```
def RGB =
  block
    red   = Token Natural
    green = Token Natural
    blue  = Token Natural

def WS = Match1 (0 | 9 | 12 | 32 | '\n' | '\r')

def Natural =
  block
    let ds = Many (1..) Digit
    ^ for (val = 0; d in ds) (addDigit val d)

def Digit =
  block
    let d = Match1 ('0' .. '9')
    ^ d - '0'

def addDigit val d =  10 * val + (d as uint 64)
```

```
-- PPM format in Daedalus
def Main =
  block
    $$ = PPM

def Token P =
  block
    $$ = P
    Many (1..) WS

def PPM =
  block
    Match "P"
    let version = Token Natural
    version == 3 is true
    width  = Token Natural
    height = Token Natural
    maxVal = Token Natural
    data   = Many height (Many width RGB)
```

Data dependency in Daedalus

The parser behaviour depends on the *width* and *height* values computed during earlier parsing

```
-- PPM format in Daedalus
def Main =
  block
    $$ = PPM

def Token P =
  block
    $$ = P
    Many (1..) WS

def PPM =
  block
    Match "P"
    let version = Token Natural
    version == 3 is true
    width  = Token Natural
    height = Token Natural
    maxVal = Token Natural
    data   = Many height (Many width RGB)
```

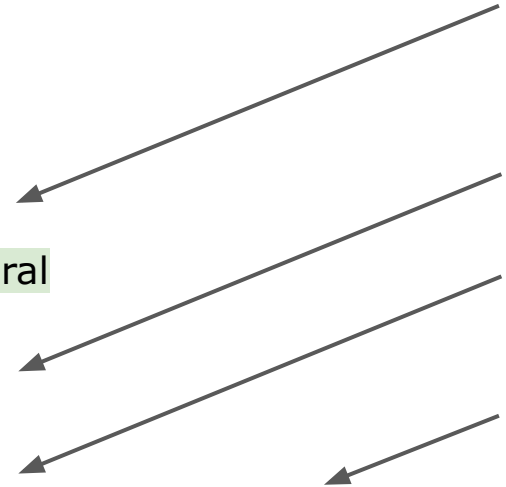
Daedalus reflects the intuitive spec:

A magic number identifying the file type (for ASCII PPM, this is P3)

The dimensions of the image (width then height)

The maximum color value

A 'matrix' of RGB triples for each pixel defined in row-major order



Try Daedalus

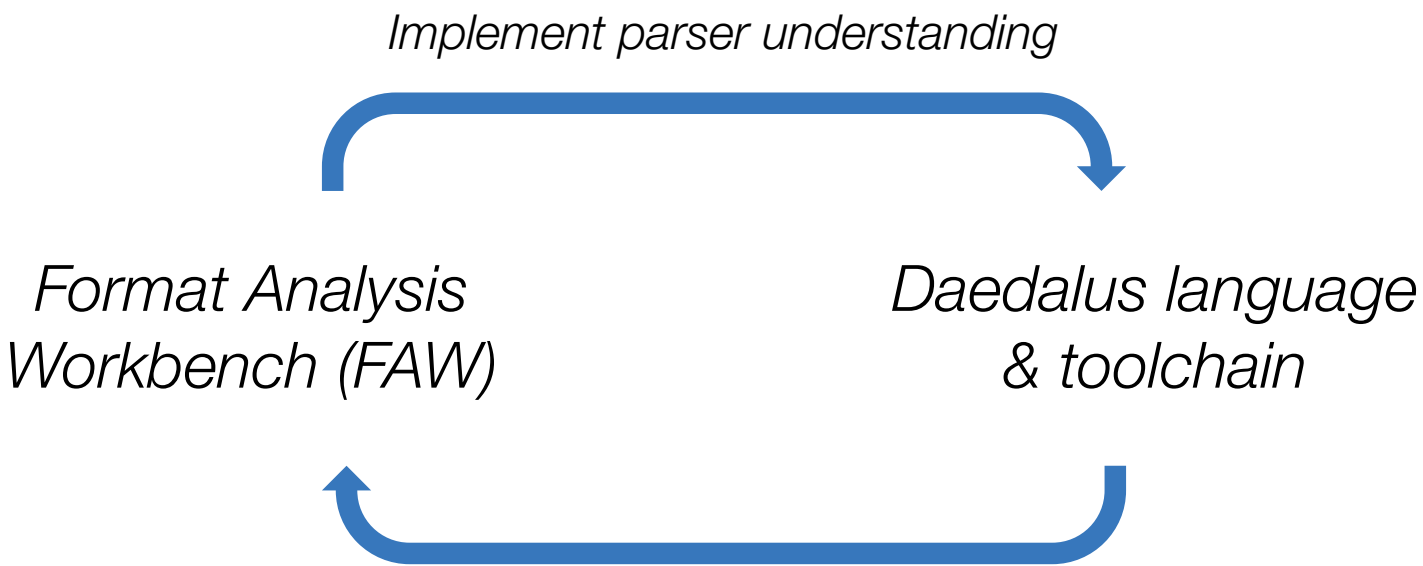
Tutorial: https://galoisinc.github.io/daedalus/tutorial/index.html

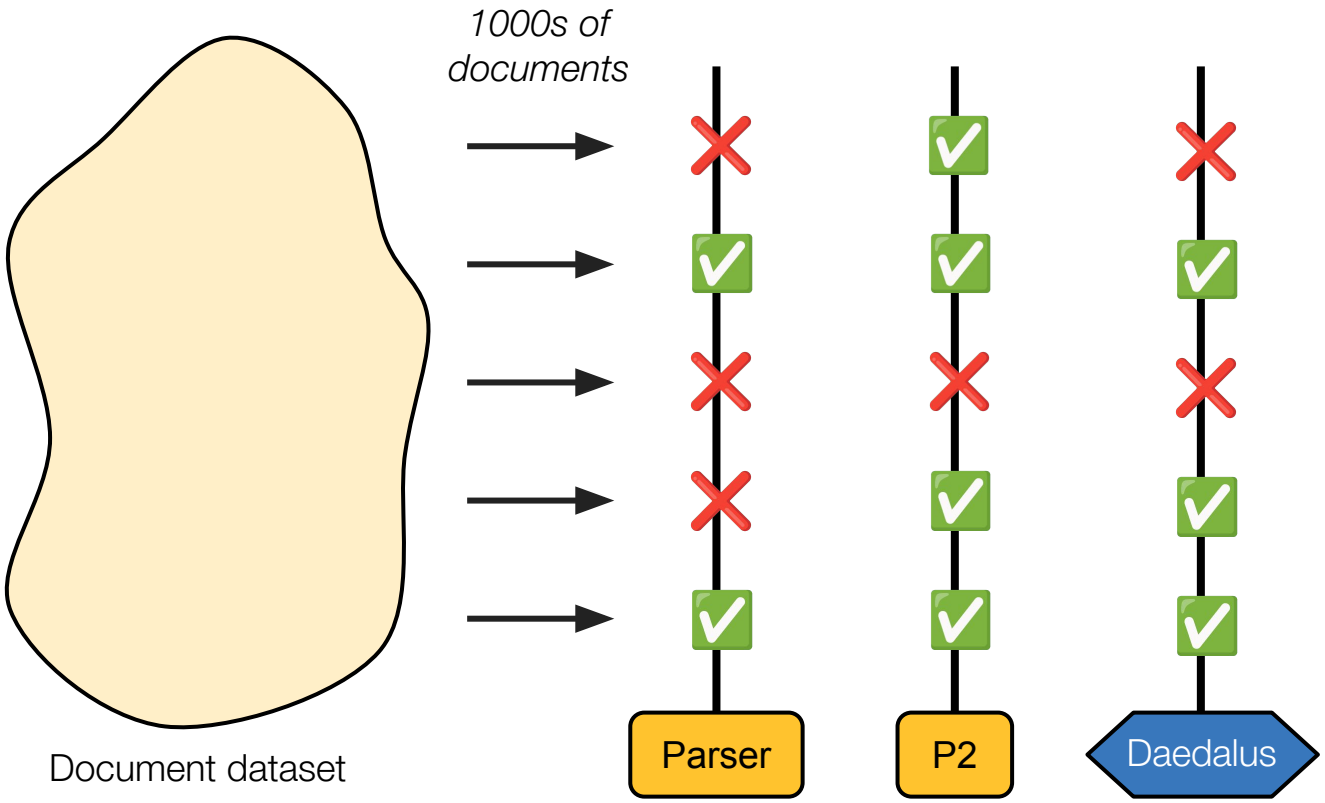
try-Daedalus, a framework for developing Daedalus in VSCode, using a remote container: https://github.com/galoisinc/try-Daedalus

FAW

\+

Daedalus


Implement parser understanding
Format Analysis Workbench (FAW)
Daedalus language & toolchain
Test new safer parsers



Parser \\ Daedalus	rejected	valid
rejected	X %	A %
valid	B %	Y %

We tested FAW + Daedalus a lot!

- Daedalus definitions and generated parsers for **14 formats (inc PDF)**
- Analyzed **13 PDF parsers, 5 NITF parsers** and **1MM+ documents**
- Discovered **9 issues with PDF specification**, **10s of bugs in parsers**
- Working with the PDF Foundation to develop a machine-readable specification of PDF that eliminates common vulnerabilities

We built other things (thanks, DARPA!)

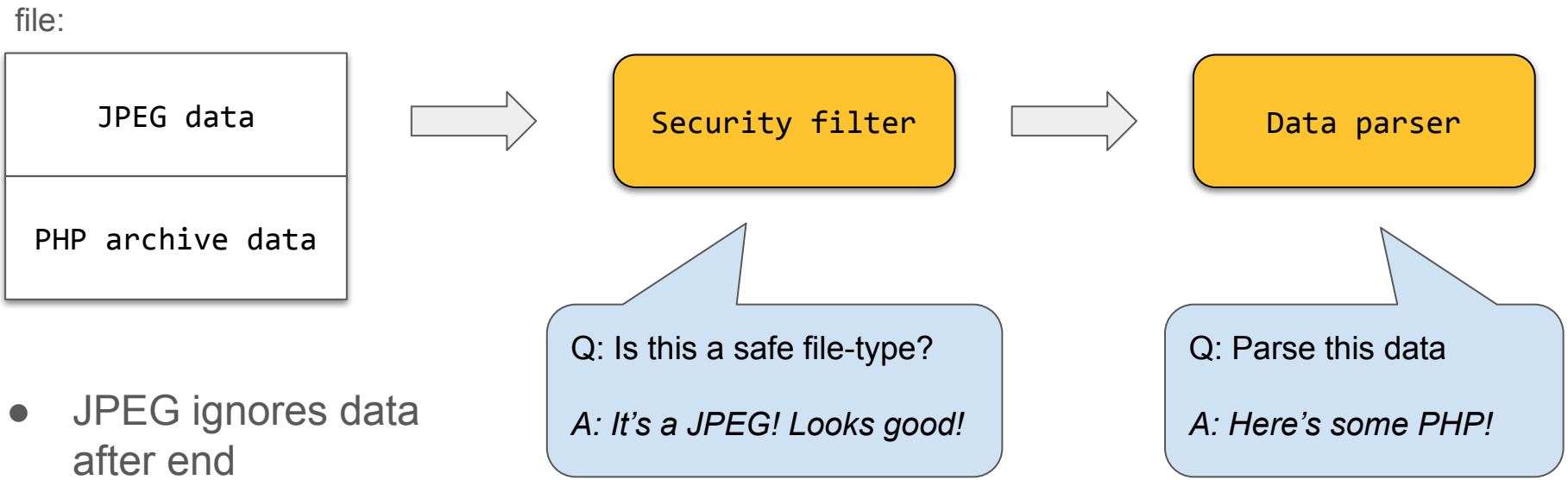
Talos, an object synthesizer based on symbolic analysis of Daedalus specs

- Daedalus: parse bytes into a semantic value
- Talos: from a format and desired semantic value, construct the input bytes

HTTP smuggling detection, FAW + Talos to find HTTP parser differentials

Polyglot detection, based on Daedalus + static analysis

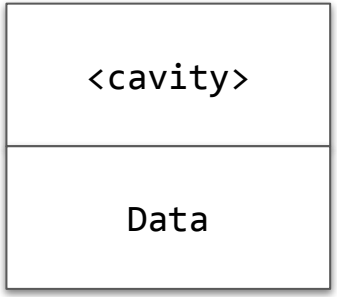
Polyglots are bad



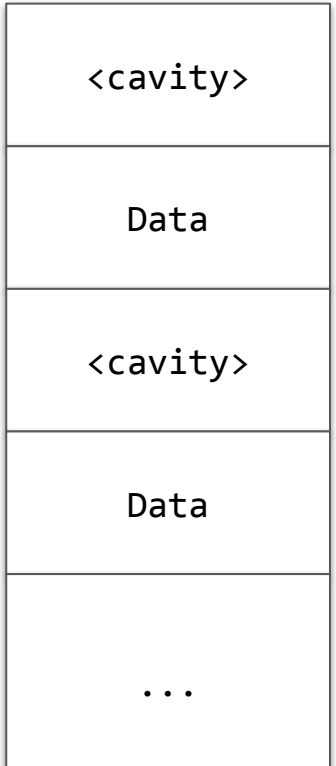
- JPEG ignores data after end
- PHP archive ignores data before 'magic' start string

Polyglots are caused by *cavities* (& other things too)

AB type:



Zipper type:



- Cavities don't affect the resulting semantic value
- Caused by eg. comments, start characters

Eg: Evan Sultanik @ Trail of Bits - resume is PDF and NES ROM:

https://www.sultanik.com/cv

Static cavity detection in formats

Cavity detection process:

- Write the format in Daedalus
- Use a context-sensitive, flow-insensitive analysis to track how parsed data is handled
- Cavities form when data is parsed but not tested by the program

Detect potential polyglots based on a Daedalus format description

Also: synthesize polyglot instances using Talos

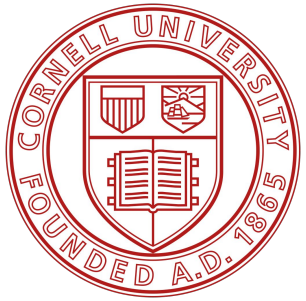
Daedalus as a target for format analysis!

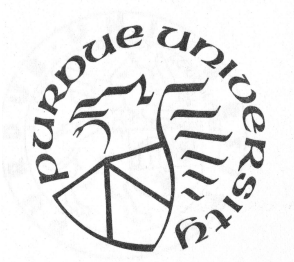
Summary:

1. Parsing matters a lot and is very hard
2. *Eg:* PDF, an interesting and horrible format
3. Two core problems in safer parsing
4. Some progress: FAW & Daedalus

SafeDocs team

- Galois (Prime)
- Trail of Bits
- RTI
- Narf Industries
- Verocel
- Cornell
- Penn State
- Princeton
- Purdue
- Tufts





Parsing is, unfortunately, still very hard

Some problems we thought about but didn’t solve:

- Parser verification, especially for extant parsers
- Subsetting / filtering parser languages
- Managing variants of formats (eg. a spec vs non-conformant versions)
- Specifying schema descriptions (eg. JSON) alongside data formats
- Synthesis of format specifications from examples

Someone should solve these problems too …

miked@galois.com

|galois|