Parsers

Daniel Wallach
DARPA
Information Innovation Office (I2O)

Workshop on Language-Theoretic Security (LangSec)

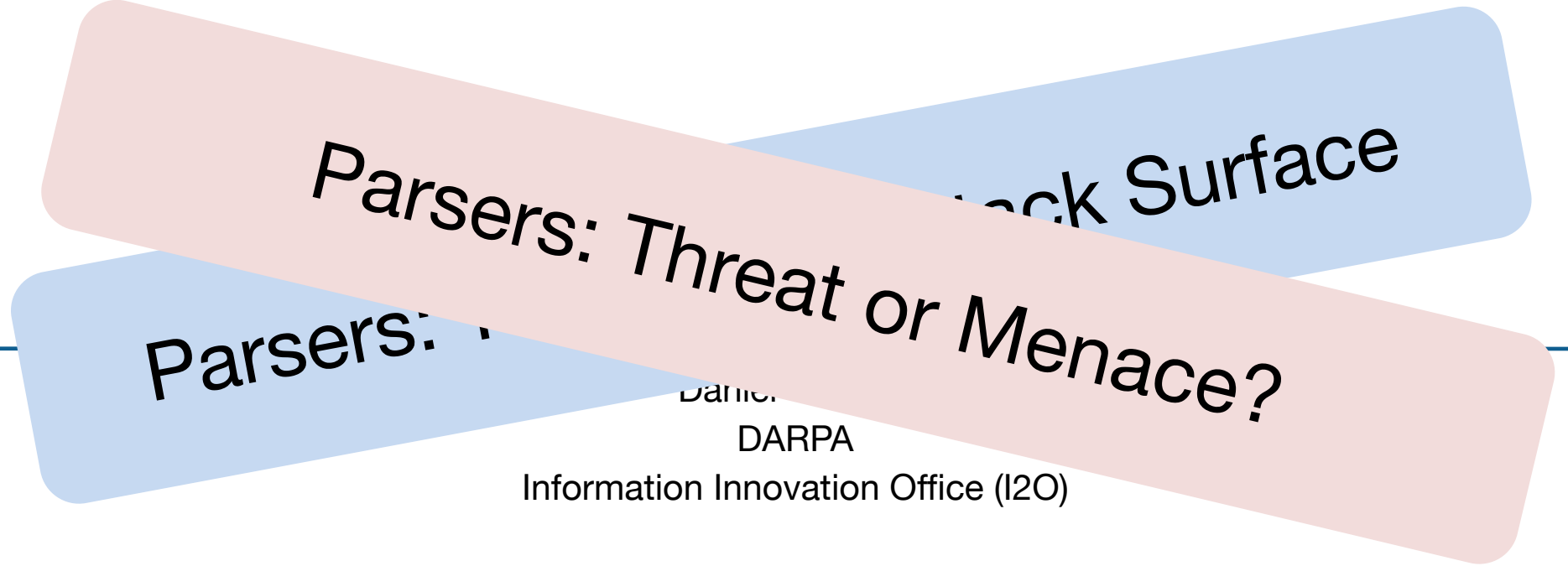
15 May 2025

Parsers: The Fractal Attack Surface

Daniel Wallach
DARPA
Information Innovation Office (I2O)

Workshop on Language-Theoretic Security (LangSec)

15 May 2025



Dan
DARPA
Information Innovation Office (I2O)

Workshop on Language-Theoretic Security (LangSec)

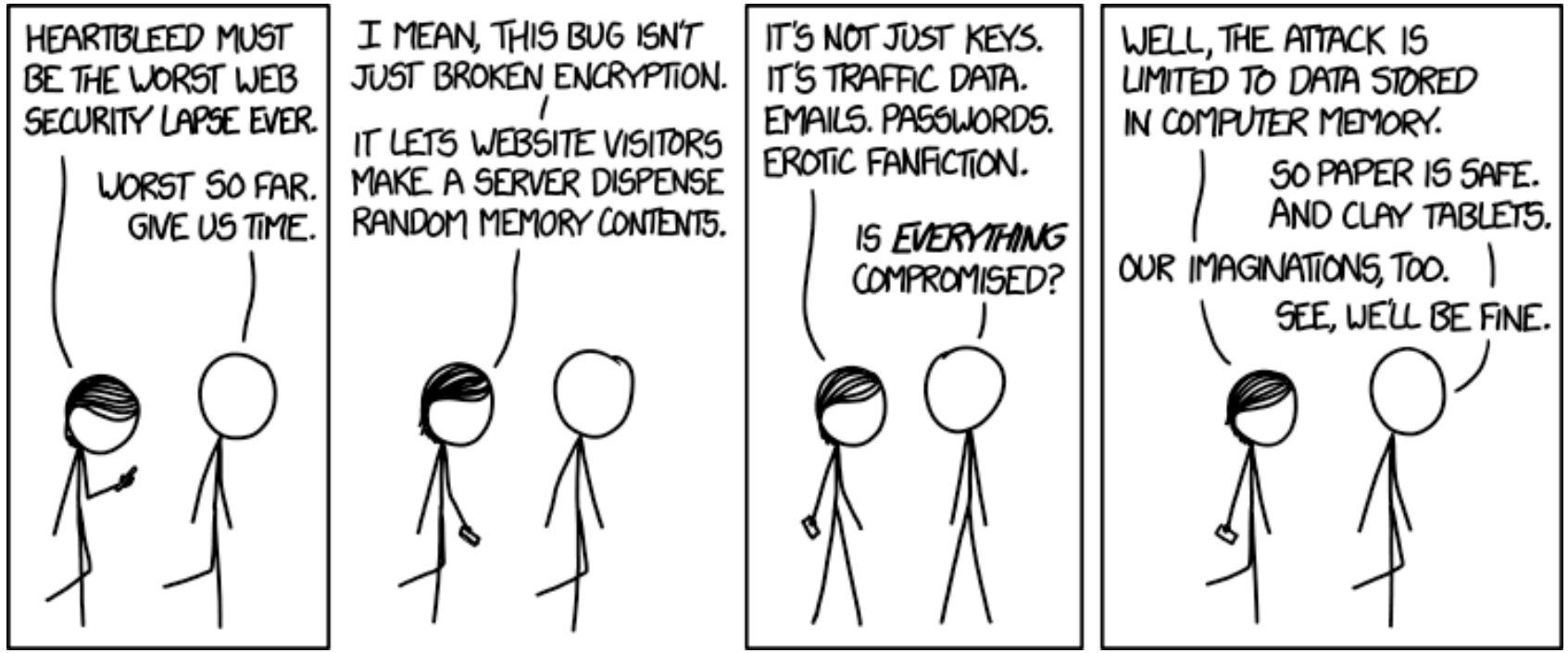
15 May 2025

What is a parser?

Parsers (computer security perspective)

- Parsers convert (potentially untrusted) bytes to (potentially sensitive) internal data structures
- Parsers are the outer edge of the attack surface of every program!

- And, in C or C++, hand-written parsers are (allegedly) the source of 80% of CVEs
  - Developers take shortcuts, make unsafe assumptions
  - Example: Heartbleed OpenSSL bug: trusting a length field to be correct → attacker can read sensitive memory



https://xkcd.com/1353/

CVE: Common Vulnerabilities and Exposures (https://www.cve.org/About/Overview)

Parsers (formal languages perspective)

- A language is a set of rules (a grammar) defined over words
  - Automata theory: Different classes of grammars (e.g., “regular” vs. “context free”) require different classes of machines to recognize them
- Words (or tokens) are defined over an alphabet
  - Lexical analysis: Rules to convert from characters to tokens (typically defined with regular expressions)
- So, what does a parser do?
  - Accept all messages inside the language
  - Reject all messages outside the language
- Sometimes lexical analysis and parsing are done in two separate phases, sometimes all at once
- Parsers don’t (traditionally) enforce higher-level rules
  - Static semantics analysis, done after parsing, enforces the rules of a programming languages (e.g., type checking)

“Colorless green ideas sleep furiously” (Noam Chomsky, 1957)
Grammatically correct text can still be semantic nonsense.

Image credit: https://hightechhistory.wordpress.com/tag/noam-chomsky/

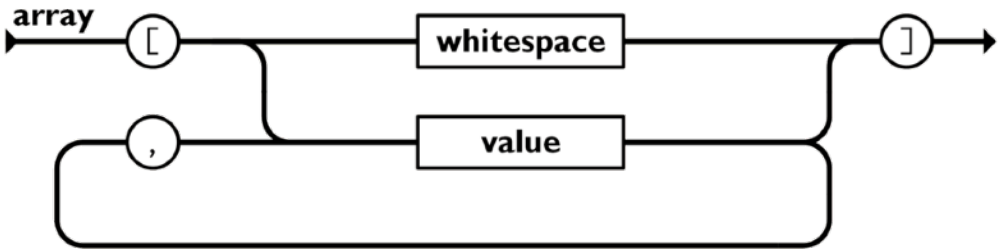
Parsers (pragmatic perspective)

- Every computer program has data structures that represent its internal state
  - Serializers convert internal data structures to external representations (bytes)
  - Deserializers convert bytes back into internal data structures
  - Other terms for this: marshalling/unmarshalling, pickling/unpickling
- Many attempts in CS history to create general-purpose serialization infrastructure
  - ASN.1 (1984) defines textual and binary ("packed") representations, used widely in telephony, cryptography
  - Google's Protocol Buffers are (in effect) a modern redo (and simplification) of ASN.1
  - Write down message definitions in interface definition language (IDL), code synthesized automatically
  - Also popular: human-readable plain-text data formats (XML, JSON, LISP S-expressions, YAML, etc.)
    - Lots of extensions, e.g., JSON schemas, to enforce some (but not all) semantic rules
- And, of course, seemingly homebrew alternatives
  - Streaming audio and video protocols (join in the middle of a stream, resync after errors)
  - Dump the in-memory representation to disk (Microsoft Office's original .doc, .xls, .ppt formats)

    https://www.joelonsoftware.com/2008/02/19/why-are-the-microsoft-office-file-formats-so-complicated-and-some-workarounds/
    - All sorts of security ramifications (e.g., "fast save" appends to the file, so old text isn't actually deleted)

Parsers (functional programming nerd perspective)

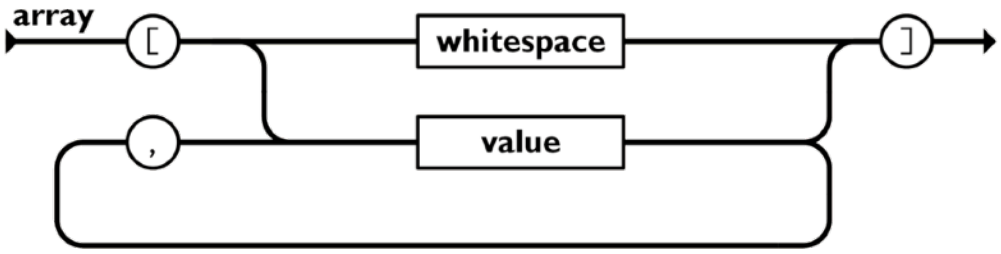
- Parsers are algebraic data types! We can combine small parsers into bigger parsers.
- Example JSON parser (written with the Angstrom parser combinator library for TypeScript)



JSON spec for an array (json.org)

Parsers (functional programming nerd perspective)

- Parsers are algebraic data types! We can combine small parsers into bigger parsers.
- Example JSON parser (written with the Angstrom parser combinator library for TypeScript)



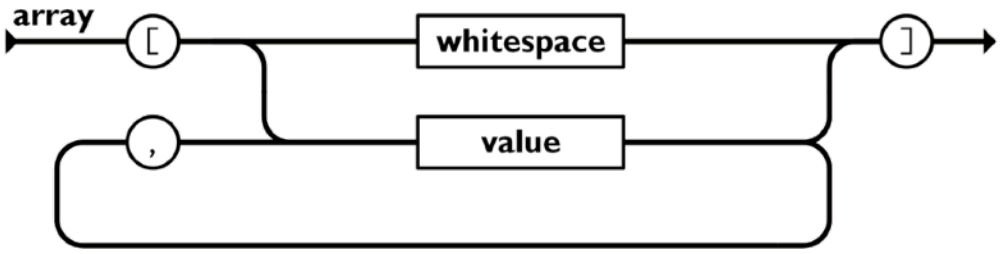
JSON spec for an array (json.org)

```
let parser =
  fix(parse => {
    let arrayParser =
      char('[') *> sep_by(comma, parse) <* char(']') >>|
        (a => Array(a));
    let member = lift2(pair, keyParser <* colon, parse);
    let objectParser =
      char('{') *> sep_by(comma, member) <* char('}') >>|
        (o => Object(o));

    peek_char_fail
    >>= (
      c =>
        switch (c) {
          | '"' => stringParser
          | 't'
          | 'f' => boolParser
          | 'n' => nullParser
          | '[' => arrayParser
          | '{' => objectParser
          | _ => numberParser
        }
    );
  });
```

https://reasonml.chat/t/a-gentle-introduction-to-parser-combinators-and-angstrom/2546

Parsers (functional programming nerd perspective)

- Parsers are algebraic data types! We can combine small parsers into bigger parsers.
- Example JSON parser (written with the Angstrom parser combinator library for TypeScript)



JSON spec for an array (json.org)

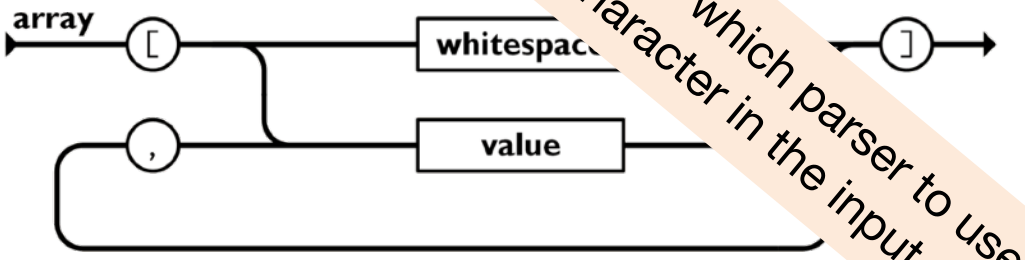
```
let parser =
  fix(parse => {
    let arrayParser =
      char('[') *> sep_by(comma, parse) <* char(']') >>|
                  Array(a));
    let m                ift2(pair, keyParser <* colon, parse);
    let                  er =
                       > sep_by(comma, member) <* char('}') >>|
                         ect(o));

                    har_fail
                  (
      c =>
        switch (c) {
          | '"' => stringParser
          | 't'
          | 'f' => boolParser
          | 'n' => nullParser
          | '[' => arrayParser
          | '{' => objectParser
          | _ => numberParser
        }
    );
  });
```

sep_by (separate by) "combines" the comma parser with the top-level value parser

https://reasonml.chat/t/a-gentle-introduction-to-parser-combinators-and-angstrom/2546

ers (functional programming nerd perspective)

- Parsers are alg a types! We can combine small parsers into bigger parsers.
- Example JSON pa n with the Angstrom parser combinator library for TypeScript)



JSON spec for an array (json.org)

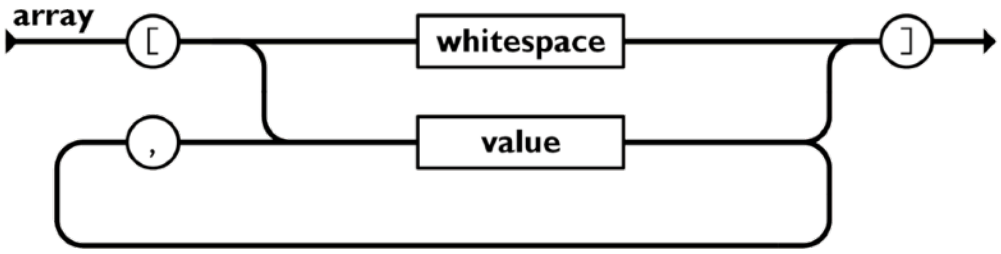
peek_char_fail decides which parser to use based on the next character in the input

```
let parser =
  fix(parse => {
    let arrayParser =
      char('[') *> sep_by(comma, parse) <* char(']') >>|
        (a => Array(a));
    let member = lift2(pair, keyParser <* colon, parse);
    let objectParser =
      char('{') *> sep_by(comma, member) <* char('}') >>|
        (o => Object(o));

    peek_char_fail
    >>= (
      c =>
        switch (c) {
          | '"' => stringParser
          | 't'
          | 'f' => boolParser
          | 'n' => nullParser
          | '[' => arrayParser
          | '{' => objectParser
          | _ => numberParser
        }
    );
  });
```

https://reasonml.chat/t/a-gentle-introduction-to-parser-combinators-and-angstrom/2546

Parsers (functional programming nerd perspective)

- Parsers are algebraic data types! We can combine small parsers into bigger parsers.
- Example JSON parser (written with the Angstrom parser combinator library for TypeScript)



JSON spec for an array (json.org)

```
let parser =
  fix(parse => {
    let arrayParser =
      char('[') *> sep_by(comma, parse) <* char(']') >>|
        (a => Array(a));
    let member = lift2(pair, keyParser <* colon, parse);
    let objectParser =
      char('{') *> sep_by(comma, member) <* char('}') >>|
        (o => Object(o));

    peek_char_fail
    >>= (
      c =>
        switch (c) {
          | '"' => stringParser
          | 't'
          | 'f' => boolParser
          | 'n' => nullParser
          | '[' => arrayParser
          | '{' => objectParser
          | _ => numberParser
        }
    );
  });
```

Parser combinator code looks (more) like an English spec.

https://reasonml.chat/t/a-gentle-introduction-to-parser-combinators-and-angstrom/2546

Parsers (functional programming nerd perspective)

- Parsers are algebraic data types! We can combine small parsers into bigger parsers.
- Example JSON parser (written with the Angstrom parser combinator library for TypeScript)

```
let parser =
  fix(parse => {
    let arrayParser =
      char('[') *> sep_by(comma, parse) <* char(']') >>|
        (a => Array(a));
    let member = lift2(pair, keyParser <* colon, parse);
    let objectParser =
      char('{') *> sep_by(comma, member) <* char('}') >>|
        (o => Object(o));

    peek_char_fail
    >>= (
      c =>
        switch (c) {
          | '"' => stringParser
          | 't'
          | 'f' => boolParser
          | 'n' => nullParser
          | '[' => arrayParser
          | '{' => objectParser
          | _ => numberParser
        }
    );
  });
```

Notably absent: error-handling code. (But it's still there under the hood.)

Parser combinator code looks (more) like an English spec.

https://reasonml.chat/t/a-gentle-introduction-to-parser-combinators-and-angstrom/2546

How hard is it to write a parser?

CVEs for JSON parsers in C and C++

“Find all the CVEs for popular C and C++ JSON parsers and categorize the vulnerabilities.”

(Gemini 2.0 Flash, “Deep Research”, May 2025. “Gemini can make mistakes, so double-check it”)

Image credit: https://vlipsy.com/vlip/spongebob-squarepants-a-little-longer-than-a-few-minutes-later-pTepukCR

CVEs for JSON parsers in C and C++

"Find all the CVEs for popular C and C++ JSON parsers and categorize the vulnerabilities."
(Gemini 2.0 Flash, "Deep Research", May 2025. "Gemini can make mistakes, so double-check it")

Image credit: https://vlipsy.com/vlip/spongebob-squarepants-a-little-longer-than-a-few-minutes-later-pTepukCR

A sampling of parser CVEs for JSON in C and C++

- Lexical analysis / tokenization vulnerabilities

CVE ID	Affected Library	Description	Severity
CVE-2016-4303	cJSON	Mishandling of UTF8/16 strings in parse_string leading to heap-based buffer overflow	Unknown
CVE-2016-10749	cJSON	Buffer over-read in parse_string when string ends with backslash	Unknown

A sampling of parser CVEs for JSON in C and C++

- Data handling and semantic interpretation vulnerabilities

CVE ID	Affected Library	Description	Severity
CVE-2024-38517	RapidJSON	Integer underflow in GenericReader::ParseNumber()	High
CVE-2024-39684	RapidJSON	Integer overflow in GenericReader::ParseNumber()	High
CVE-2023-26819	cJSON	Denial of service via crafted JSON document with a large number	Low

A sampling of parser CVEs for JSON in C and C++

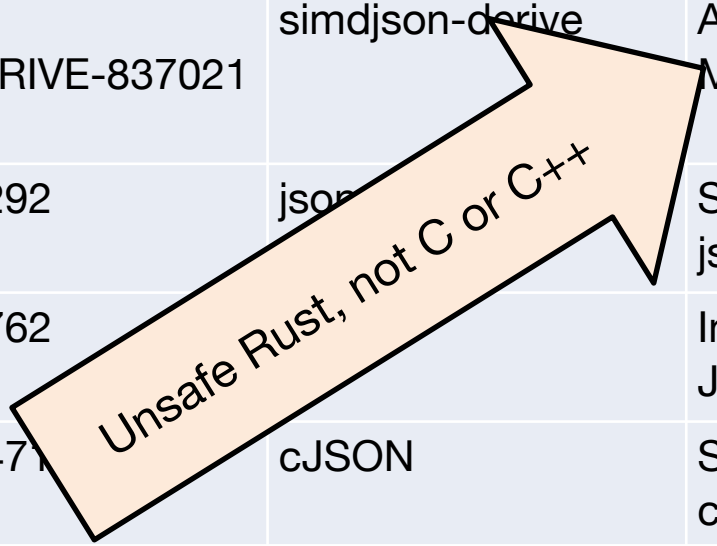
- Memory management vulnerabilities

CVE ID	Affected Library	Description	Severity
SNYK-UNMANAGED-NLOHMANNJSON-6387367	nlohmann/json	Heap-based buffer overflow during CBOR parsing due to unclosed UTF-8 string	High
CVE-2019-15550	simdjson	Out-of-bounds read and incorrect crossing of a page boundary	High
SNYK-RUST-SIMDJSONDERIVE-8370210	simdjson-derive	Access of uninitialized pointer due to misuse of MaybeUninit (Rust)	High
CVE-2021-32292	json-c	Stack-buffer-overflow in parseit function of json_parse sample program	Critical
CVE-2020-12762	json-c	Integer overflow and out-of-bounds write via large JSON file	Unknown
CVE-2023-50471	cJSON	Segmentation violation via cJSON_InsertItemInArray	High
CVE-2023-50472	cJSON	Segmentation violation via cJSON_SetValuestring	Unknown
CVE-2024-31755	cJSON	Segmentation violation via cJSON_SetValuestring	Unknown

A sampling of parser CVEs for JSON in C and C++

- Memory management vulnerabilities

CVE ID	Affected Library	Description	Severity
SNYK-UNMANAGED-NLOHMANNJSON-6387367	nlohmann/json	Heap-based buffer overflow during CBOR parsing due to unclosed UTF-8 string	High
CVE-2019-15550	simdjson	Out-of-bounds read and incorrect crossing of a page boundary	High
SNYK-RUST-SIMDJSONDERIVE-8370210	simdjson-derive	Access of uninitialized pointer due to misuse of MaybeUninit (Rust)	High
CVE-2021-32292	json	Stack-buffer-overflow in parseit function of json_parse sample program	Critical
CVE-2020-12762		Integer overflow and out-of-bounds write via large JSON file	Unknown
CVE-2023-5047[illegible]	cJSON	Segmentation violation via cJSON_InsertItemInArray	High
CVE-2023-50472	cJSON	Segmentation violation via cJSON_SetValuestring	Unknown
CVE-2024-31755	cJSON	Segmentation violation via cJSON_SetValuestring	Unknown



7			
CVE-2019-15550	simdjson	Out-of-bounds read and incorrect crossing of a page boundary	High
SNYK-RUST-SIMDJSONDERIVE-8370210	simdjson-derive	Access of uninitialized pointer due to misuse of MaybeUninit (Rust)	High
CVE-2021-32292	json-c	Stack-buffer-overflow in parseit function of json_parse sample program	Critical
CVE-2020-12762	json-c	Integer overflow and out-of-bounds write via large JSON file	Unknown
CVE-2023-50471	cJSON	Segmentation violation via cJSON_InsertItemInArray	High
CVE-2023-50472	cJSON	Segmentation violation via cJSON_SetValuestring	Unknown
CVE-2024-31755	cJSON	Segmentation violation via cJSON_SetValuestring with NULL argument	Unknown
CVE-2018-1000217	cJSON	Use After Free vulnerability	High
CVE-2019-11834	cJSON	Out-of-bounds access related to multiline comments	Unknown
CVE-2019-11835	cJSON	Out-of-bounds access related to \x00 in string literal	Unknown

A sampling of parser CVEs for JSON in C and C++

- Error handling and input validation vulnerabilities

CVE ID	Affected Library	Description	Severity
CVE-2024-38525	dd-trace-cpp (using nlohmann/json)	Uncaught exception when logging malformed unicode	High
CVE-2024-34363	Envoy (using nlohmann/json)	Uncaught exception when serializing incomplete UTF-8 strings	High
CVE-2019-1010239	cJSON	Null dereference in cJSON_GetObjectItemCaseSensitive() due to improper condition check	High
AIKIDO-2024-10263	JsonCpp	Out-of-bounds read in getLocationLineAndColumn during error message generation	Low

A sampling of parser CVEs for JSON in C and C++

- Denial of service

CVE ID	Affected Library	Description	Severity
CVE-2013-6401	Jansson	Predictable hash collisions leading to denial of service	Medium

Rice Comp215 (“Introduction to Program Design”)

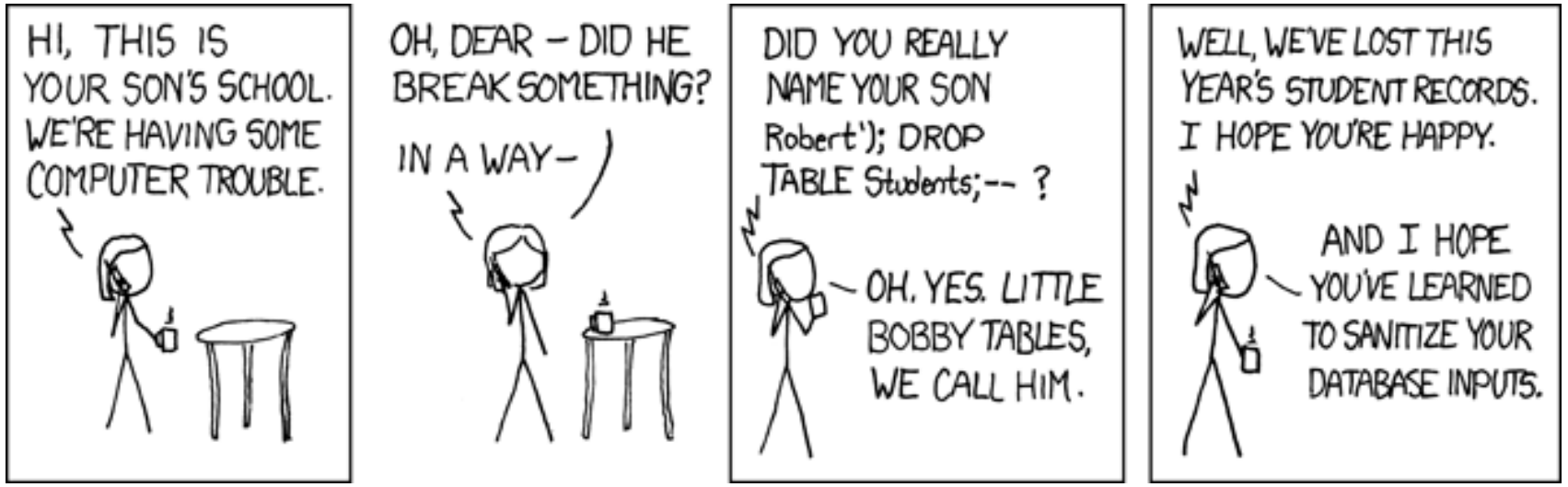
- Required sophomore CS class at Rice, introduces Java programming
  - Taught by me, 2014-2019
- Two-week student assignment: write a JSON parser (week 1: tokenize, week 2: recursive parsing)
- Observations & bugs:
  - I provided string escaping/unescaping; Apache Commons String library failed a simple fuzz test
  - Java’s regular expression engine ran out of memory matching individual strings greater than 10KB
    - Flex (lexical analysis, code synthesis tool) worked correctly
  - Some students would look ahead more than one token: Slow and incorrect!
  - Sophisticated tests (property-based testing / fuzzing) helped students fix their bugs
  - Subtle details matter
    - Not every float can be represented in JSON (NaN, +/-Infinity)
    - JSON can express big integers; should we support Java’s BigInteger class?
  - Undefined by JSON: what should happen if you see the same key twice in a JSON object?

- In subsequent weeks, students had to write code to convert data to/from JSON
  - JSON deserialization requires manual checks for data semantics

But Java is a safe programming language!

- Memory-safe programming languages (roughly, everything but C and C++) guarantee your code will behave in a deterministic, well-defined way
  - E.g., Reading beyond the end of an array is defined to fail predictably, rather than be a security attack vector
  - Memory safety would have defeated the Heartbleed vulnerability (and many, many others)

- A buggy parser, even in a safe language, can still be bad for security
  - Security decisions are made based on the outputs of parsers
  - Code injection attacks (cross-site scripting, SQL injection) can be viewed as attacks on parsers



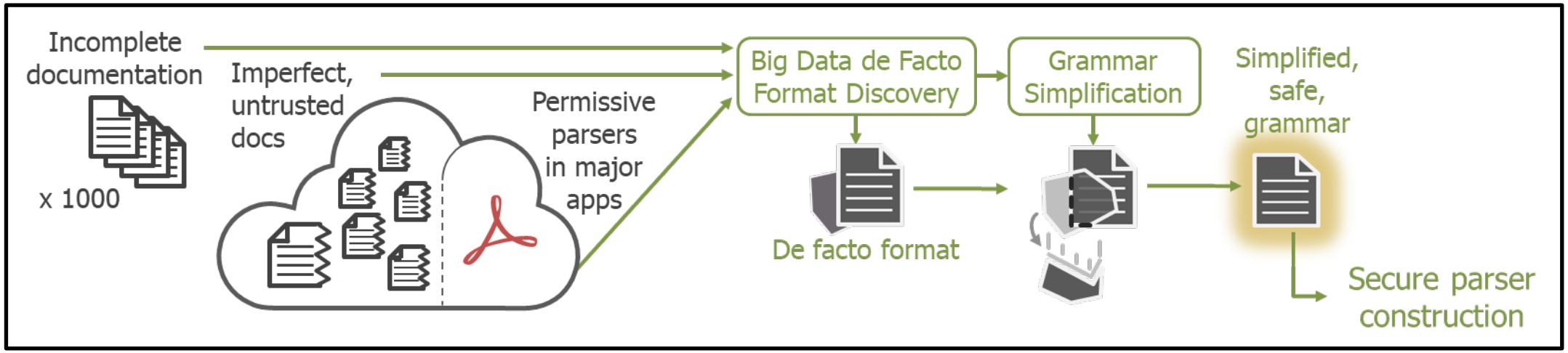
https://xkcd.com/327/

SafeDocs: Hardened parsers for legacy software

Why Safe Documents (SafeDocs)?

Precise format definitions -- We need them here, everywhere, yesterday!

- As descriptions of **interfaces**, it's critical that the definitions of data formats be **explicit**, **machine-readable**, and **unambiguous**
- But in practice, **they're not**: e.g., PDF ISO standard is **984 pages**, with **100+ ambiguities** found by the SafeDocs program alone
- Secondary consequence: With no specification, it is impossible to **verify parsers**
- Problem has been cleanly defined since **computing's antiquity**, so why isn't it actually solved?
  - One extreme: **Context free grammars** are well understood, but can't describe actual formats
  - Other extreme: **Parser combinators** are powerful but don't shield a (possibly non-programmer) format expert from creating unsafe semantic actions (which are needed to validate formats!)



SafeDocs objective met?

Hypothesis: Formal methods are effective for defining and safely ingesting secure parsers for complex and widely deployed real-world formats with divergent implementations

- SafeDocs researchers developed new methods and tools to allow people to trust what they see on their screens and to click confidently on documents
- SafeDocs advanced the state of the art in verification of the security of data format parsers and eliminated the primary source of preventable, parsing vulnerabilities
- **SafeDocs program enabled a huge step towards the DARPA vision of a world without software vulnerabilities**
- Hypothesis was met

SafeDocs impact: The Arlington PDF Model (named after DARPA site)

- First vendor-neutral, open-source, specification-derived, machine and human-readable definition of PDF objects (across all version of the standard 1.6-2.0)

Arlington PDF model

PDF association

First **open access**, **vendor neutral**, **specification-derived**, **machine readable** definition of **every** PDF 2.0 object

Set of 515 text-based TSV files -- Single PDF object per TSV

- Structured data: 12 columns with custom predicates
  - >3,500 rows
  - >1700 assertions using 39 unique predicates
- No-code accessible:
  - EBay “big data” **tsv-utilities**, Linux CLI
- Low-effort programmatic consumption
  - Python, C++, Java
- Validated against a vendor proprietary model and >10^6 files of extant data

SafeDocs: **100+** issues discovered in **ISO 32000-2** (PDF 2.0), over 600 issues in SoTA PDF software

#	Column Name
1	Key name / array index
2	Type
3	Since Version
4	Deprecated In
5	Required?
6	Indirect Reference?
7	Inheritable?
8	Default Value
9	Possible Values
10	Special Case
11	Link
12	Notes (*freeform text*)

https://github.com/pdf-association/arlington-pdf-model

Document complexity leads to vulnerabilities

Users of past PDF standard had to deal
with 1,000 pages of this mess

Every key in every dictionary.
For arrays, every array element

All basic PDF COS types, plus a few others for convenience

Version introduced (*unstated = PDF 1.0*). Version deprecated (*where appropriate*)

Table 31 — Entries in a page object

Key	Type	Value
Type	name	(*Required*) The type of PDF object that this dictionary describes; shall be *Page* for a page object or *Template* for an invisible Template page (see 12.7.7, "Named pages").
Parent	dictionary	(*Required; shall be an indirect reference*) The page tree node that is the immediate parent of this page object. Objects of Type *Template* shall have no **Parent** key.
LastModified	date	(*Required if* ***PieceInfo*** *is present; optional otherwise*; *PDF 1.3*) The date and time (see 7.9.4, "Dates") when the page's contents were most recently modified. If a page-piece dictionary (**PieceInfo**) is present, the modification date shall be used to ascertain which of the application
AF	array of dictionaries	(*Optional*; *PDF 2.0*) An array of one or more file specification dictionaries (7.11.3, "File specification dictionaries") which denote the associated files for this page. See 14.13, "Associated files" and 14.13.8, "Associated files linked to DParts" for more details.
OutputIntents	array	(*Optional*; *PDF 2.0*) An array of output intent dictionaries that shall specify the colour characteristics of output devices on which this page might be rendered (see 14.11.5, "Output intents").
DPart	dictionary	(*Required if this page is within the range of a DPart, not permitted otherwise*; *PDF 2.0*) An indirect reference to the DPart dictionary whose range of pages includes this page object (see 14.12.3, "Connecting the

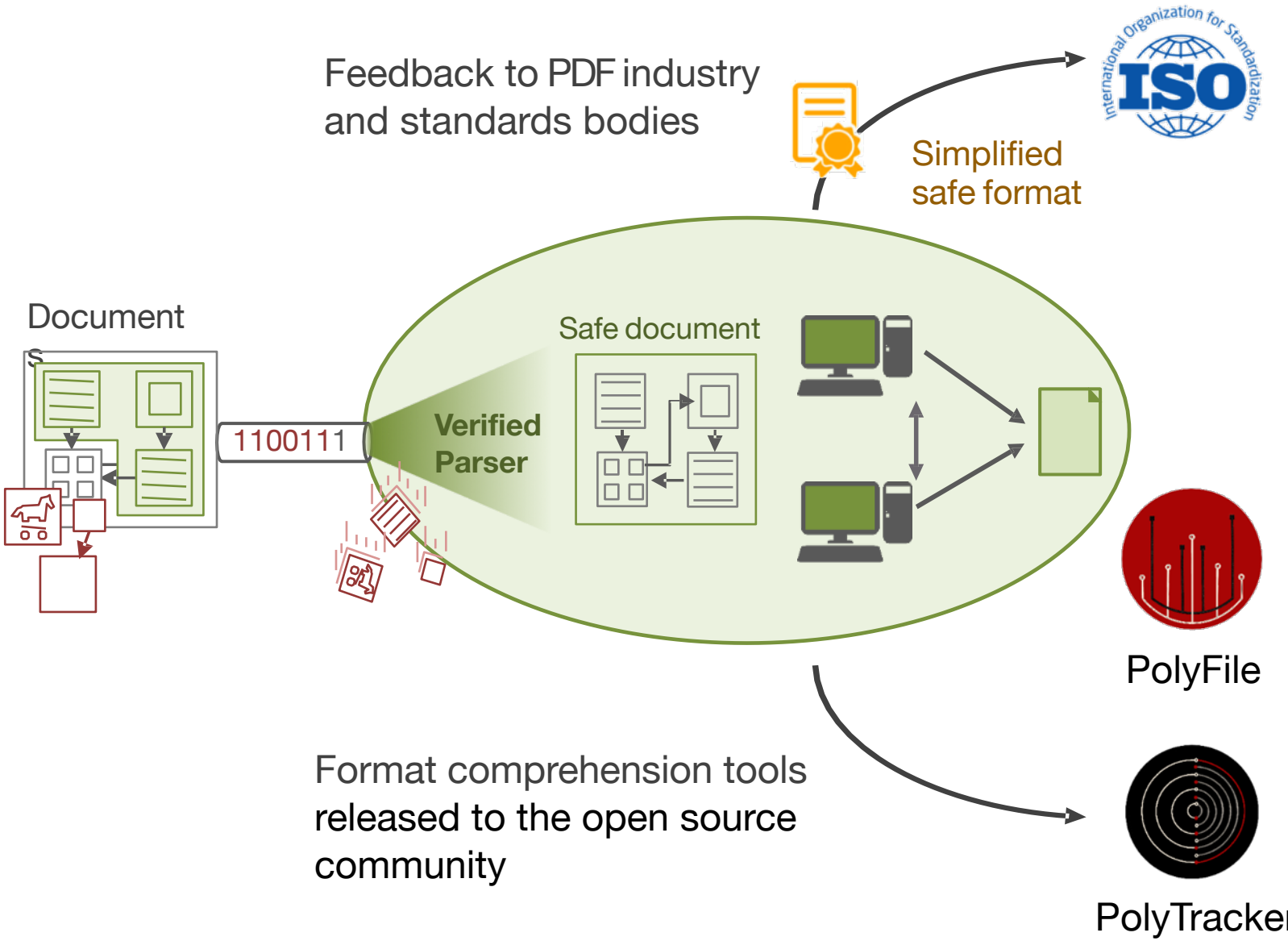
Indirect & Direct requirements

Required-ness / Optional, with conditions

Linkage to precise type(s) for each key value when dictionary or array

SafeDocs approach: Prose → structured data

SafeDocs tools targeting the electronic documents community



SafeDocs identified and submitted fixes for **100+** ambiguities in the ISO 32000-2 PDF standard that are a source of vulnerabilities:

- "Frankenstein" objects that allow ambiguous interpretation [information hiding]
- Excessive object indirection, ambiguities in dictionary object structure (e.g., indirect keys) [parser exploitation, detection evasion]
- Ambiguities in allowed object nesting (e.g., streams in arrays) [parser exploitation]

First dedicated tool for exploring polyglot and "schizophrenic" file phenomena

- Deep inspection of a file's bytes
- Extensible, based on TrID and KaiTai struct data definitions

First dedicated tool for intelligent tracing of parsers written in C/C++

- Instruments a parser to output a map from each input byte to parser functions
- Scales to real parsers, via novel data flow tracking technique

SafeDocs highlights: Data Definition Languages (DDL), format models

(1)

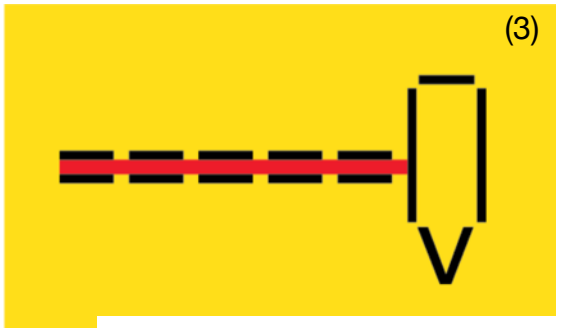
Daedalus (Galois, Inc.)

(2)

Parsley

Parsley DDL (SRI)

(3)



Parser combinators for binary formats, in C. Yes, in C. What? Don't look at me like that.

Hammer/VALARIN

(4)

(Special Circumstances/ Riverside Research)

Arlington PDF Model

(PDF Association)

Image credits:
[1] https://en.wikipedia.org/wiki/Icarus#/media/File:Gowy-icaro-prado.jpg
[2] Natarajan Shankar, SRI
[3] Meredith L. Patterson, Special Circumstances LLC
[4] PDF Association

Daedalus

- A framework for defining and parsing practical formats, consisting of
  - An expressive Data Definition Language (DDL)
  - A high-assurance parser generator
- Used to define and parse ~14K lines of practical formats
  - PDF, IccMAX, National Imagery Transmission Format (NITF), Data Distribution Service (DDS), Micro Air Vehicle Link (MAVLink), ...
- Parser-generator targets **C++** and Haskell, depending on parser client and constraints on performance
  - Implements an efficient **ownership-based** memory manager
  - Latest PDF/NITF parsers have been tested on billions of documents/10^3 of CPU hours with 0 errors found
- Implements Language Server Protocol (LSP): Can be written, type-checked, and tested in Visual Studio Code (**VSCode)**, Editing MACros (Emacs), ...

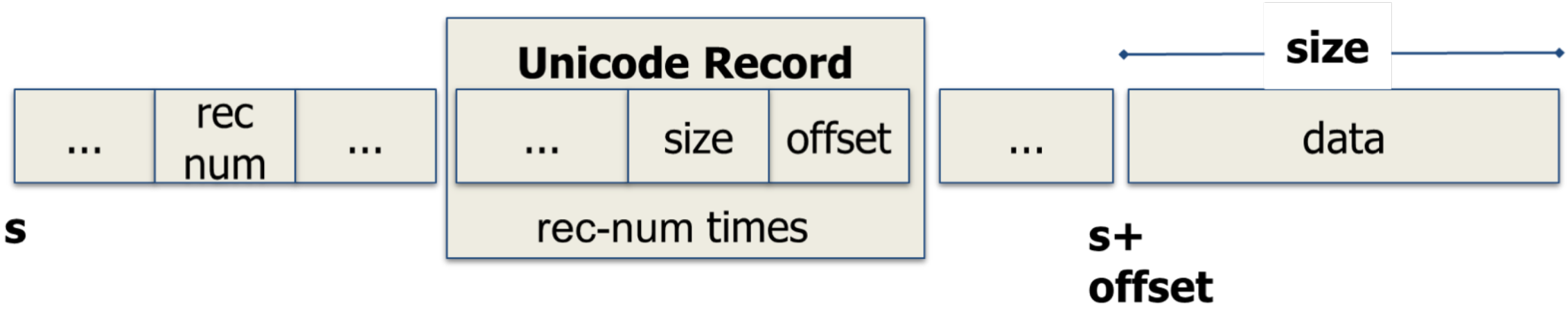
Key features: Succinct definitions with **higher-order parsers**, precise definitions with **data-dependent binds**, practical definitions with **first-class input streams** (i.e., streams can be bound and parsed multiple times)

Available at github.com/GaloisInc/daedalus

ICC color profiles and IccMAX in Daedalus

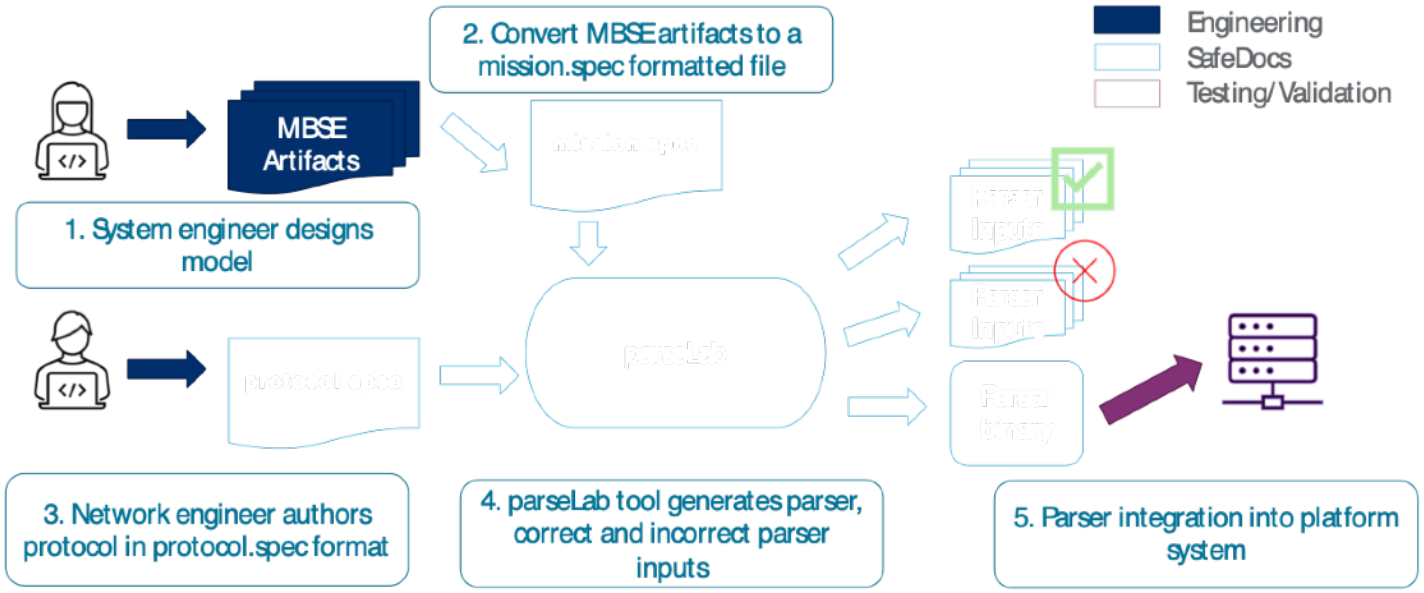
- Language of **color profiles**: Translations between color spaces (e.g., RGB or CMYK)
  - Standardized as ISO 20677:2019 standard for image technology color management across major operating systems, medical imaging, high-resolution imagery
  - 2020 flaw in Android's ICC profile handling disabled phones when a flaw-triggering image was set as background -- SafeDocs explicated the root cause of the bug:

    https://www.riverloopsecurity.com/blog/2020/07/android-systemui-icc/
- Worked with the PDF Association to define format in **537 LOC**
- Key technical challenge: Format specified using **complex stream arithmetic**
  - Can be defined in **~10 lines** in DaeDaLus!



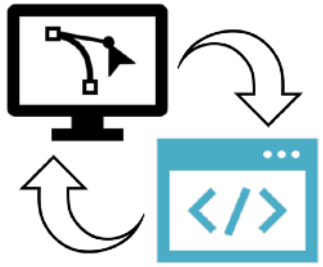
SafeDocs ParseLab (Lockheed Martin Advanced Technology Labatories)

- What is it?
  - ParseLab is a modular framework for generating protocol parsers as well as inputs necessary to validate and test generated parsers
- How can ParseLab be used?
  - Generate syntactic parsers for protocol messages
  - Generate invalid and valid binary packet data based on the specification (specification guided fuzzing)
  - Generate unit tests to validate generated modules
- What do I have to specify to use ParseLab?
  - Protocol specification file with message fields, data types and constraints
  - Parser toolkit generator module (Hammer supported)

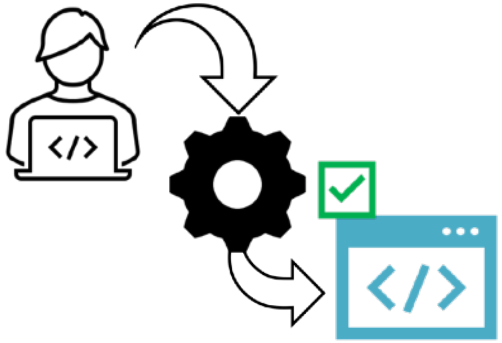


ParseLab is a tool that enables rapid parser development, data generation and validation

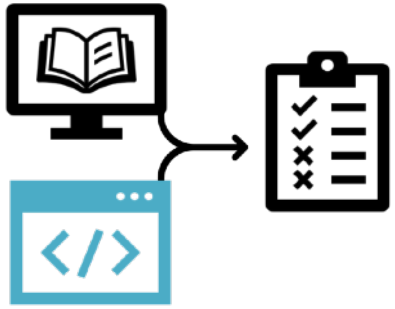
SafeDocs' ParseLab tool motivation for transition to the defense industrial base



Bridge the design implementation gap between system design and parser development



Generation of secured parsers using Model-Based Systems Engineering (MBSE) tools without the need for expertise in formal methods



Validate parser implementations against specified syntax and semantics

parseLab allows system engineers to create secure parsers without needing formal methods expertise

SafeDocs transition effort

- Currently expanding the capabilities of **parseLab** and **Hammer** for a DoD transition partner to **provide resilient parsing** of binary protocol messages within a platform systems of systems

- Extension of these capabilities include:

  - **Legacy sensor hardening:** Augmenting the authentication process for legacy platform sensors using their protocols (such as X11) with deep-message validation from secure parsers generated from Google Protocol Buffers (GPB) .PROTO specifications
  - **Systems engineering transition:** Reducing the gap between design and implementation by integrating systems engineering tools (e.g., Cameo) and protocol parser generators to enable specification of constraints and semantics of system interactions to **auto-generate** secure parsers -- thereby reducing the gap between design and implementation
  - **Support for operational requirements:** Expanding parser generation to include parsers for GPB on-the-wire binary format and validating the serialized data to include constraints without deserializing the data first to support the ubiquitous usage of GPB-encoded data throughout platform systems

Focus on maturing features for use with an operational platform release

How does Hammer work? It's another parser combinator!

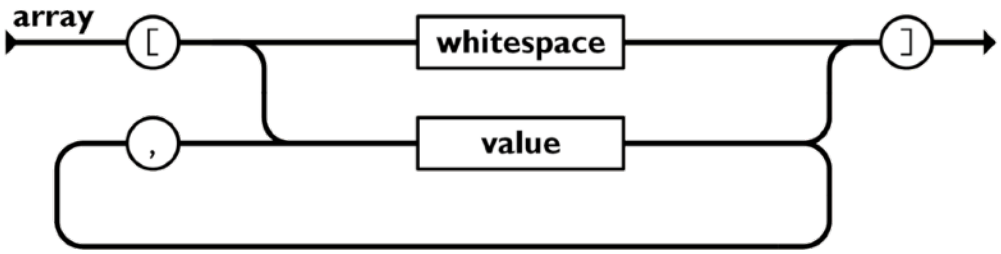
- But it's in C, tuned for performance
- Works for binary and text formats
- ParseLab takes high-level input, emits Hammer code

```
void init_parser() {
  /* Whitespace */
  H_RULE(ws, h_in((uint8_t*)" \r\n\t", 4));
  /* Structural tokens */
  H_RULE(left_square_bracket,
    h_middle(h_many(ws), h_ch('['), h_many(ws)));
  H_RULE(right_square_bracket,
    h_middle(h_many(ws), h_ch(']'), h_many(ws)));
  H_RULE(comma, h_middle(h_many(ws), h_ch(','), h_many(ws)));

…

  /* Forward declarations */
  HParser *value = h_indirect();

…

  /* Arrays */
  H_ARULE(json_array,
    h_middle(left_square_bracket,
      h_sepBy(value, comma),
      right_square_bracket));
```

https://github.com/sergeybratus/HammerPrimer/blob/master/lecture_13/json.c

How does Hammer work? It's another parser combinator!

- But it's in C, tuned for performance
- Works for binary and text formats
- ParseLab takes high-level input, emits Hammer code



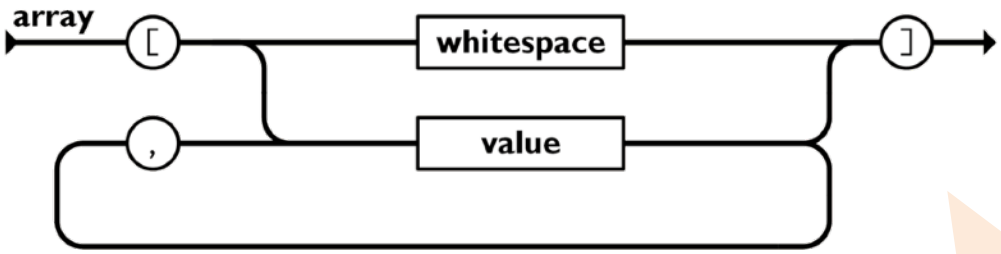
JSON spec for an array (json.org)

```
void init_parser() {
  /* Whitespace */
  H_RULE(ws, h_in((uint8_t*)" \r\n\t", 4));
  /* Structural tokens */
  H_RULE(left_square_bracket,
    h_middle(h_many(ws), h_ch('['), h_many(ws)));
  H_RULE(right_square_bracket,
    h_middle(h_many(ws), h_ch(']'), h_many(ws)));
  H_RULE(comma, h_middle(h_many(ws), h_ch(','), h_many(ws)));

…

  /* Forward declarations */
  HParser *value = h_indirect();

…

  /* Arrays */
  H_ARULE(json_array,
    h_middle(left_square_bracket,
      h_sepBy(value, comma),
      right_square_bracket));
```

https://github.com/sergeybratus/HammerPrimer/blob/master/lecture_13/json.c

How does Hammer work? It's another parser combinator!

- But it's in C, tuned for performance
- Works for binary and text formats
- ParseLab takes high-level input, emits Hammer code



JSON spec for an array (json.org)

```
void init_parser() {
  /* Whitespace */
  H_RULE(ws, h_in((uint8_t*)" \r\n\t", 4));
  /* Structural tokens */
  H_RULE(left_square_bracket,
    h_middle(h_many(ws), h_ch('['), h_many(ws)));
  H_RULE(right_square_bracket,
    h_middle(h_many(ws), h_ch(']'), h_many(ws)));
  H_RULE(comma, h_middle(h_many(ws), h_ch(','), h_many(ws)));

…

  /* Forward declarations */
  HParser *value = h_indirect();

…

  /* Arrays */
  H_ARULE(json_array,
    h_middle(left_square_bracket,
      h_sepBy(value, comma),
      right_square_bracket));
```

h_sepBy (separate by) combinator, same as we saw earlier

https://github.com/sergeybratus/HammerPrimer/blob/master/lecture_13/json.c

And, while we're at it, how about nom in Rust?

- No surprise, more parser-combinators
- Macros could have made this easier for humans to write, but they pulled macros out in Nom 5.0 for better debugging & compiler performance
- Broad thoughts
  - Parser-combinators seem to be a crowd favorite
  - But they don't help much with static semantics checks

```rust
fn array<'a, E: ParseError<&'a str> + ContextError<&'a str>>(
  i: &'a str,
) -> IResult<&'a str, Vec<JsonValue>, E> {
  context(
    "array",
    preceded(
      char('['),
      cut(terminated(
        separated_list0(preceded(sp, char(',')), json_value),
        preceded(sp, char(']')),
      )),
    ),
  )
  .parse(i)
}
```

https://github.com/rust-bakery/nom/blob/main/examples/json.rs

And, while we're at it, how about nom in Rust?

- No surprise, more parser-combinators
- Macros could have made this easier for humans to write, but they pulled macros out in Nom 5.0 for better debugging & compiler performance
- Broad thoughts
  - Parser-combinators seem to be a crowd favori
  - But they don't help much with static sema checks

```
fn array<'a, E: ParseError<&'a str> + ContextError<&'a str>>(
  i: &'a str,
) -> IResult<&'a str, Vec<JsonValue>, E> {
  context(
    "array",
    preceded(
      char('['),
      cut(terminated(
        separated_list0(preceded(sp, char(',')), json_value),
        preceded(sp, char(']')),
      )),
    )
    .  rse(i)
}
```

`separated_list0`, same as we saw earlier

https://github.com/rust-bakery/nom/blob/main/examples/json.rs

Getting fancier: Daedalus

- Daedalus defines its own functional programming language for writing parsers
  - Concise syntax for doing parser combinators
  - Provably safe output synthesized for C++ or Haskell
- Some really interesting features
  - Parsers can reason about non-local data (e.g., table of contents with offsets for actual data)
  - Support for eager or lazy parsing, streaming
  - Static semantics rules are just Daedalus code
- Runtime performance: sometimes 3-5x faster than other parser generators!
- Externally red-teamed PDF parser
- Bonus feature: Talos uses the Daedalus rules to synthesize valid inputs (you get a fuzzer for free)

```
def JSON_value =
  First
    Null    = JSON_null
    Bool    = JSON_bool
    Number  = JSON_number
    String  = JSON_string
    Array   = JSON_array_of JSON_value
    Object  = JSON_object_of JSON_value

def JSON_array_of P =
  block
    $['[']
    let buf =
         case Optional (JSON_ws_then P) of
           nothing -> builder
           just v  -> emit builder v
    $$ = build
           (many (buf = buf)
              block
                JSON_ws_then $[',']
                emit buf (JSON_ws_then P)
           )
    JSON_ws_then $[']']
```

https://dl.acm.org/doi/pdf/10.1145/3656410 (PLDI 2024 paper)

Getting fancier: Daedalus

- Dae… …tional programming language for w…
  - Concise syntax for doing…
  - Provably safe output synthesized f…
- Some really interesting features
  - Parsers can reason about non-local data (e.g., table… contents with offsets for actual data)
  - Support for eager or lazy parsing, streaming
  - Static semantics rules are just Daedalus code
- Runtime performance: sometimes 3-5x faster than other parser generators!
- Externally red-teamed PDF parser
- Bonus feature: Talos uses the Daedalus rules to synthesize valid inputs (you get a fuzzer for free)

Parsers passed as arguments to parsers! Note: recursive definitions are allowed.

```
def JSON_value =
  First
    Null    = JSON_null
    Bool    = JSON_bool
    Number  = JSON_number
    String  = JSON_string
    Array   = JSON_array_of JSON_value
    Object  = JSON_object_of JSON_value

def JSON_array_of P =
  block
    $['[']
    let buf =
         case Optional (JSON_ws_then P) of
           nothing -> builder
           just v  -> emit builder v
    $$ = build
           (many (buf = buf)
              block
                JSON_ws_then $[',']
                emit buf (JSON_ws_then P)
           )
    JSON_ws_then $[']']
```

https://dl.acm.org/doi/pdf/10.1145/3656410 (PLDI 2024 paper)

Getting fancier: Daedalus

- Daedalus defines its own functional programming language for writing parsers
  - Concise syntax for doing parser combinators
  - [illegible]afe output synthesized for C++ or Haskell
- S[illegible] features
  - Parse[illegible] data (e.g., table of contents with [illegible]
  - Support for eager or lazy p[illegible]
  - Static semantics rules are just Daeda[illegible]
- Runtime performance: sometimes 3-5x faster tha[illegible] other parser generators!
- Externally red-teamed PDF parser
- Bonus feature: Talos uses the Daedalus rules to synthesize valid inputs (you get a fuzzer for free)

First a JSON value, or nothing at all

```
def JSON_value =
  First
    Null    = JSON_null
    Bool    = JSON_bool
    Number  = JSON_number
    String  = JSON_string
    Array   = JSON_array_of JSON_value
    Object  = JSON_object_of JSON_value

def JSON_array_of P =
  block
    $['[']
    let buf =
         case Optional (JSON_ws_then P) of
           nothing -> builder
           just v  -> emit builder v
    $$ = build
           (many (buf = buf)
              block
                JSON_ws_then $[',']
                emit buf (JSON_ws_then P)
           )
    JSON_ws_then $[']']
```

https://dl.acm.org/doi/pdf/10.1145/3656410 (PLDI 2024 paper)

Getting fancier: Daedalus

- Daedalus defines its own functional programming language for writing parsers
  - Concise syntax for doing parser combinators
  - Provably safe output synthesized for C++ or Haskell
- Some really interesting features
  - Parsers can reason about non-local data (e.g., table of contents with offsets for actual data)
  - Support for eager or lazy parsing, streaming
  - Static semantics rules are just Daedalus code
- Runtime performance: sometimes 3-5x faster than other parser generators!
- Externally red-teamed PDF parser
- Bonus feature: Talos uses the Daedalus rules to synthesize valid inputs (you get ...

```
def JSON_value =
  First
    Null    = JSON_null
    Bool    = JSON_bool
    Number  = JSON_number
    String  = JSON_string
    Array   = JSON_array_of JSON_value
    Object  = JSON_object_of JSON_value

def JSON_array_of P =
  block
    $['[']
    let buf =
         case Optional (JSON_ws_then P) of
           nothing -> builder
           just v  -> emit builder v
    $$ = build
           (many (buf = buf)
              block
                JSON_ws_then $[',']
                emit buf (JSON_ws_then P)
           )
    JSON_ws_then $[']']
```

Then, zero or more blocks, each with a comma then a JSON value

https://dl.acm.org/doi/pdf/10.1145/3656410 (PLDI 2024 paper)

Work in progress: semi-automatic parser extraction

- DARPA V-SPELLS project: sophisticated whole-program source and binary code analysis tools
- Phase 3 challenge (ongoing): aid the developer to extract a parser from an existing program
  - Even if the codebase has parsing logic spread out across many locations (“shotgun parser”)
- Wouldn’t it be nice to help the developer replace their shotgun parser with something robust?
- The V-SPELLS BAE & Purdue team asked a related question: could we use this to find parser bugs?

V-SPELLS: Verified Security and Performance Enhancement of Large Legacy Software
https://www.darpa.mil/research/programs/verified-security-and-performance-enhancement-of-large-legacy-software

Investigating 5G network protocols

- What is 5G
  - The fifth generation mobile network standard.
  - Supports faster data rates, low latency, and massive device connectivity.
  - Powers enhanced mobile broadband, IoT, and mission-critical communications
- Layers of Protocols in 5G
  - Radio Access Network (RAN) Protocols (between devices and base stations), e.g., NR, RRC, …
  - Core Network Protocols (5G Core - 5GC), i.e., NGAP, SCTP, PFCP, and GTP
  - Security & Authentication Protocols, e.g., 5G-AKA, IPSec,…
- Open5Gs
  - Open-source 5G Core implementation (5GC & EPC)
  - 2,000+ GitHub stars, 800+ forks, used widely in research and testing

NGAP	NG Application Protocol
SCTP	Stream Control Transmission Protocol
PFCP	Packet Forwarding Control Protocol
GTP	GPRS Tunnelling Protocol (GPRS = General Packet Radio Service)
AKA	Authentication and Key management
S1AP	S1 Application Protocol (S1 = 4G interoperation mode)
5GC	5G Core
EPC	Evolved Packet Core

Open5Gs Core Protocols Are Complex

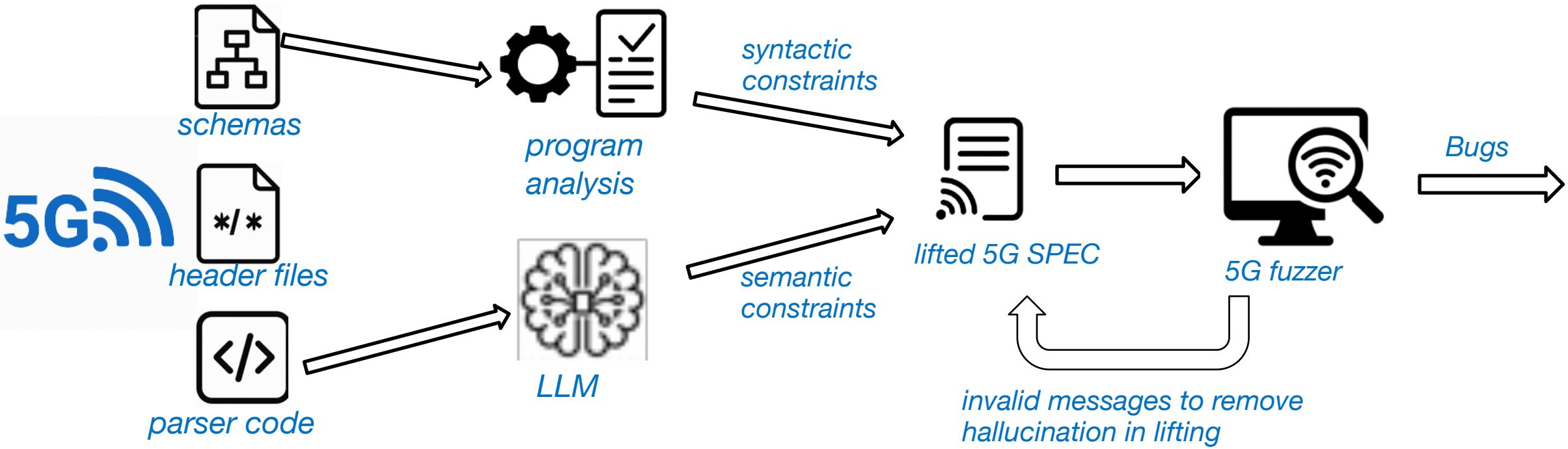
Protocols	RFC pages	Parsing Source Files	Schema Files	Field Definition Lines	Message Types
NGAP	469	153	2135	/	120
S1AP	379	153	1336	/	98
PFCP	389	43	/	7184	25
GTP-v2	414	32	/	4509	31

- Each RFC may refer to many other RFCs
- TCP has only 8 message types, which share the same header; in contrast, 5G core has close to 300 message types
- Heavily using set of unordered elements, which are in the form of Type-Length-Value (TLV)
- Heavily use CHOICE (similar to Union in C)
- Each message type has on average 20 variants, i.e., 120*20=2400 variants for NGAP

NGAP	NG Application Protocol
SCTP	Stream Control Transmission Protocol
PFCP	Packet Forwarding Control Protocol
GTP	GPRS Tunnelling Protocol (GPRS = General Packet Radio Service)
AKA	Authentication and Key management
S1AP	S1 Application Protocol (S1 = 4G interoperation mode)
5GC	5G Core
EPC	Evolved Packet Core

RFC: Request for Comments (standards document)

Neural-Symbolic Approach in Lifting and Checking



Zero-Day Bugs Found in Open5Gs

Protocols	Spec pages	Parsing Files	Schema Files	Item Definition Lines	Message Types	Bugs	Confirmed
NGAP	469	153	2135	/	120	3	2
S1AP	379	153	1336	/	98	2	2
PFCP	389	43	/	7184	25	21	21
GTP-v2	414	32	/	4509	31	16	16

- 37 bugs in PFCP and GTPV2 cause stack/heap overflow, integer overflow or assertion failure, resulting in either arbitrary code execution or DDoS attack that crash the server.
  - The root cause is that when parsing a sub field in the message, the program lacks the validity check. For example, it misses the length check of the sub field and directly call *memcpy*, causing overflow.

- 5 functional bugs are in NGAP (3) and S1AP (2) that trigger assertion, resulting in DDoS attack that crash the server. The root cause is the incorrect order of packet received.

NGAP	NG Application Protocol
SCTP	Stream Control Transmission Protocol
PFCP	Packet Forwarding Control Protocol
GTP	GPRS Tunnelling Protocol (GPRS = General Packet Radio Service)
AKA	Authentication and Key management
S1AP	S1 Application Protocol (S1 = 4G interoperation mode)
5GC	5G Core
EPC	Evolved Packet Core

www.darpa.mil