Daedalus: Safer Document Parsing

IAVOR S. DIATCHKI, Galois, USA
MIKE DODDS, Galois, USA
HARRISON GOLDSTEIN, University of Pennsylvania, USA
BILL HARRIS, Galois, USA
DAVID A. HOLLAND, Galois, USA
BENOIT RAZET, Galois, USA
COLE SCHLESINGER, Galois, USA
SIMON WINWOOD, Galois, USA

Despite decades of contributions to the theoretical foundations of parsing and the many tools available to aid in parser development, many security attacks in the wild still exploit parsers. The issues are myriad—flaws in memory management in contexts lacking memory safety, flaws in syntactic or semantic validation of input, and misinterpretation of hundred-page-plus standards documents. It remains challenging to build and maintain parsers for common, mature data formats.

In response to these challenges, we present Daedalus, a new domain-specific language (DSL) and toolchain for writing safe parsers. Daedalus is built around functional-style parser combinators, which suit the rich data dependencies often found in complex data formats. It adds domain-specific constructs for stream manipulation, allowing the natural expression of parsing noncontiguous formats. Balancing between expressivity and domain-specific constructs lends Daedalus specifications simplicity and leaves them amenable to analysis. As a stand-alone DSL, Daedalus is able to generate safe parsers in multiple languages, currently C++ and Haskell.

We have implemented 20 data formats with Daedalus, including two large, complex formats—PDF and NITF—and our evaluation shows that Daedalus parsers are concise and performant. Our experience with PDF forms our largest case study. We worked with the PDF Association to build a reference implementation, which was subject to a red-teaming exercise along with a number of other PDF parsers and was the only parser to be found free of defects.

CCS Concepts: • **Software and its engineering → Parsers**; **Functional languages**; **Domain specific languages**; • **Theory of computation → Parsing**; **Program analysis**.

Additional Key Words and Phrases: Format definition languages, binary data formats, PDF, NITF

ACM Reference Format:
Iavor S. Diatchki, Mike Dodds, Harrison Goldstein, Bill Harris, David A. Holland, Benoit Razet, Cole Schlesinger, and Simon Winwood. 2024. Daedalus: Safer Document Parsing. *Proc. ACM Program. Lang.* 8, PLDI, Article 180 (June 2024), 25 pages. https://doi.org/10.1145/3656410

Authors' addresses: Iavor S. Diatchki, diatchki@galois.com, Galois, Portland, OR, USA; Mike Dodds, miked@galois.com, Galois, Portland, OR, USA; Harrison Goldstein, hgo@seas.upenn.edu, University of Pennsylvania, Philadelphia, PA, USA; Bill Harris, bll.hrris@gmail.com, Galois, Portland, OR, USA; David A. Holland, dholland@galois.com, Galois, Portland, OR, USA; Benoit Razet, benoit.razet@galois.com, Galois, Portland, OR, USA; Cole Schlesinger, coles@galois.com, Galois, Portland, OR, USA; Simon Winwood, sjw@galois.com, Galois, Portland, OR, USA.

ACM 2475-1421/2024/6-ART180
https://doi.org/10.1145/3656410

1 INTRODUCTION

Parsers are the first line of defense against software exploits. A well-written parser will transform bits into semantically-meaningful data structures, enforce syntactic and semantic well-formedness conditions, and guard against pathological input. Conversely, a poorly written parser may fail to reject malicious input or, worse, introduce new vulnerabilities in its implementation.

Parsers are also well studied. Decades of investigation have firmly established a language-theoretic foundation [2, 14, 30], and there are many tools, libraries, and frameworks available for writing parsers in practice [21, 41, 43, 54, 61]. Yet despite these advances, parsers remain a source of vulnerabilities for popular data formats. A brief survey of the NIST National Vulnerability Database at the time of this writing shows 1,319 CVEs related to parsers, including 14 high-severity and 3 critical vulnerabilities discovered in the first three months of 2023.

Indeed, building (or maintaining) a parser for an established data format poses significant challenges for developers. Consider the Portable Document Format (PDF) [42]. Developed by Adobe in 1993, PDF is now standardized as ISO 32000 and has become ubiquitous. An aspiring parser developer must contend with a complex specification—it is hundreds of pages of English text with tens of normative references to other standards documents, which are themselves quite lengthy—as well as a complex, *implicit* specification arising from the behavior of existing PDF readers, which often accept common but noncompliant format deviations for business or technical reasons.

The format itself introduces additional challenges. PDF documents are encoded using a noncontiguous, layered binary format. One layer describes a generic representation of object values, while another schema-like layer ascribes meaning to the objects. A PDF parser must support stream manipulation, to look up object references and jump to their definitions, and also validate generic values against their schemas. This second point deserves attention. One might hope to separate validation from parsing and test the well-formedness of the result once parsing completes. For PDF, this is not an option. Dependencies in the format mean that some components of the document must be parsed *and validated* before it is clear how to parse others. PDF may be particularly complex, but other established formats pose similar challenges, especially noncontiguous binary formats. Existing parser frameworks typically cannot cleanly encode such data dependencies. Instead they rely on a mix of ad hoc mechanisms that make building correct parsers difficult.

Introducing Daedalus. To address these challenges, we present the design and implementation of *Daedalus*, a domain-specific language (DSL) for specifying complex, noncontiguous binary formats and automatically generating correct, memory-safe parser implementations. The design of the Daedalus language prioritizes clarity of expression while also supporting the complex syntactic and semantic validation requirements found in standards documents.

The Daedalus language uses parser combinators to support rich data dependencies, enabling schema processing over general objects, including algebraic datatypes, maps, and arrays, as well as simpler cases like lengths. First-class stream operators naturally support noncontiguous data formats. A foreign-function interface allows invoking libraries in the target language, such as for cryptography. Daedalus is built on a core monadic language. It provides a range of notations for expressing parsing idioms naturally, such as iteration, character classes, and applicative notation.

We also present `daedalus`, an open-source implementation of Daedalus that compiles specifications to memory-safe C++ and Haskell; offers an interpreter mode that provides error traces and debug messages to assist in parser development; and comes equipped with a Language Server Protocol (LSP) [51] server for editor integration. Our evaluation shows that Daedalus specifications are concise in comparison with other tools and that `daedalus` produces performant code.

We have used `daedalus` (the tool) to generate parsers for 20 data formats, including PDF and NITF [66]. The PDF implementation was evaluated in a red-team exercise, where no vulnerabilities

were detected in the parser, although 5 defects were discovered in the handwritten C++ wrapper that invoked the parser. Competing tools were found to have tens of defects. As another point of comparison, three monthly releases of the Poppler PDF library [56] in 2022 included "Fix crashes on malformed files" in their change log, and at least two of these received CVE assignments [19, 20]. Poppler has been actively developed for over 15 years and is available on most Linux distributions.

Finally, we also briefly present two analyses built atop Daedalus as evidence of amenability to analysis: document synthesis and cavity detection. Document synthesis automatically generates valid inputs for Daedalus parsers. This assists in direct testing of parser implementations. It also assists evaluation against other tools targeting the same standard, to detect deviations from specified behavior as well as missing de facto behavior common to the data format. Cavity detection evaluates a data format for regions that can be exploited to execute code smuggling attacks.

Contributions. In summary, the contributions of this paper are:

- Daedalus, a DSL designed for building high-assurance parsers for complex binary formats.
- `daedalus`, an implementation that generates parsers in multiple languages (C++ and Haskell), with an interpreter mode for development and support for error traces and debug messages.
- An evaluation showing that Daedalus specifications are concise when compared to similar tools, the output code is performant, and Daedalus parsers have relatively fewer defects.
- Two static analyses built atop Daedalus, demonstrating amenability to analysis while assisting developers in evaluating their parsers with respect to the broader format ecosystem.

The remainder of the paper is structured as follows. Section 2 explores the complexities of PDF as motivation, then Sections 3–7 present the design and rationale for Daedalus. Section 8 shows examples of PDF in Daedalus. Section 9 discusses the compiler and other facets of the implementation. Section 10 presents the evaluation, Section 11 discusses analyzability, Section 12 covers related work, and Section 13 concludes.

2 MOTIVATING EXAMPLE: PDF

We developed Daedalus to parse complex real-world binary formats. Our main challenge-problem was *PDF* [42], which is is a mature format, used in many different ways, and with multiple implementations. It is quite complex and has many features that challenge traditional parsing and we expect *this will be true for many mature document formats*. Daedalus is designed to help solve these problems in practice, and we have validated our design by building a full-scale PDF parser. The result has been tested on hundreds of thousands of PDF documents, has been subject to extensive red-teaming with zero discovered vulnerabilities, and has uncovered multiple bugs that have been fixed in the PDF standard.

Header
```
%PDF-1.5
%âãÏÓ
```
Linearization

Body
```
9 0 obj
3.14159 % Comment
endobj

12 0 obj
(Hello world)
endobj
...
```
Cross-reference table
```
xref
0 312
0000000000 65535 f
0000000016 00000 n
0000003753 00000 n
0000002673 00000 n
...
```
Trailer
```
trailer
«
/Root 1 0 R
/Size 32
»
startref
43167
%%EOF
```

Fig. 1. PDF layout, from [65] pg. 2

A previous workshop paper [65] explored PDF's design in detail, and briefly discusses a prototype parser. Figure 1, from that paper, illustrates PDF's structure. Our work focuses on the design of Daedalus as a language, and so we draw attention only to PDF's most significant challenges.

At the highest level, a PDF document is a set of *objects* represented in the Carousel Object System (COS). One or more *cross-reference tables* serve to index these objects' locations. Cross-reference tables may be partly or wholly superseded by later tables in the document, allowing the file to be

updated by appending rather than overwriting. A superseded reference still points somewhere; the old object remains as 'garbage' in the file. Thus, identifying the valid location of an object requires first parsing a sequence of cross-reference tables at different offsets in the file.

This generates *Challenge 1*: very non-linear parsing of the document. This is compounded by other factors. An object may be contained as a compressed stream inside another object. An object's size may also be contained in another object. As a result, parsing a single object may involve parsing, retrieving and processing multiple objects across the PDF file.

PDF COS objects are constrained by a schema language, analogous to XML / JSON schemas. Unfortunately, the PDF schema cannot be fully separated from the parser. For example, cross-reference streams are a special kind of object—analogous to the cross-reference table—that contain information about the location of other objects. Examples like these lead to *Challenge 2*: very rich data dependency. To perform the schema processing, the parser must rely on the generic COS objects, which are data structures such as maps and arrays. This contrasts with simpler formats, where data dependency mostly involves integer values; for example, array lengths.

As mentioned above, COS objects may be contained in other objects. These *stream objects* are particularly difficult because they are typically compressed or encrypted. This means that a parser must be able to pause parsing, apply some computational transformation on some bytes, and then parse the transformed results. In other words, *Challenge 3* is that the parser must be able to compute over a value generated during parsing, then parse the result of this computation.

The Daedalus language design is intended to address these challenges:

(1) To allow non-linear parsing of the document, Daedalus supports an explicit notion of a stream value which can be manipulated by the grammar. This also helps address Challenge 3: streams constructed by decompression/decryption can be parsed in a first-class manner.
(2) To handle rich data dependency, Daedalus is designed with a generic notion of data dependency inherited from functional programming languages. This means that Daedalus programs are able to depend on any datatype, while allowing parsers to be cleanly structured.
(3) To allow offloading non-parsing logic, Daedalus includes a foreign function interface (FFI). For example, one might call an external library to decrypt a data stream before parsing it.

At a high level, our PDF parser is structured as follows:

- Daedalus specifications for parsing objects in the COS format;
- Daedalus specifications for processing specific PDF types—typically these happen in two steps: first we parse a generic COS object, and then we validate it and translate it into a semantic value that represents the specific PDF type;
- Custom support functions provided by the host application, including the caching mechanism described next, wrappers around OpenSSL for decryption, and object stream body decoders (FlateDecode, LZWDecode, ASCIIHexDecode, ASCII85Decode, and DCTDecode).

Processing of a PDF starts off in the application code. After creating an empty object cache, we locate and parse the cross-reference tables, by invoking a parser specified in Daedalus. Then, we call another Daedalus parser to process the document's trailer, which is the "root" of a PDF.

While parsing a PDF type, if we need to resolve a reference to a PDF object, we call back into the application code. This code consults the object cache to see if the object has already been processed. If not, it consults the cross-reference table to find the object, parse it, and store the result in the cache. Part of this logic also takes care to avoid cycles while resolving references. This system gives both good performance, due to the caching, and also a convenient way for processing PDF objects, as resolving a reference is just a non-terminal in the grammar wrapping the external call.

Note that it is possible to structure the parser in other ways. For example, we could have omitted the object cache, and parsed objects multiple times, but this has performance drawbacks. The object

cache could also be made explicit in the Daedalus specification. While possible, this adds clutter as many productions have to take the cache as an additional input and return it as an output and, arguably, the cache is an application-specific optimization.

Besides resolving references, the application code exports libraries to the Daedalus specification. In particular, when processing PDF streams we often need to transform the body of the stream before parsing. It would be somewhat impractical to specify this in Daedalus itself—typical transformations include decompression and decryption. Instead we use standard libraries, such as OpenSSL.

3 DAEDALUS BASICS

A specification in Daedalus consists of a collection of *modules*, each of which contains a collection of *declarations*. We distinguish between three kinds of declarations:

- *parser* declarations define new grammar productions,
- *semantic function* declarations define pure functions that manipulate semantic values, and
- *character classes* define predicates over single elements of the input (i.e., bytes).

The language is statically typed, using an off-the-shelf Hindley-Milner type system [58] with qualified types [49], which supports parametric polymorphism and type inference. We do not include typing judgments for the language constructs, as they are quite lengthy and not very illuminating. A few specific properties of the type system, including the use of type qualifiers as inference constraints, are discussed in Section 5.

Parser declarations are at the heart of a Daedalus specification, as they specify the strings that are accepted by the language and the semantic values that correspond to them. Our design is heavily influenced by monadic parser combinators [34], so parsers are constructed compositionally, by combining existing parsers into more complex parsers.

3.1 Primitive Parsers

At its core, Daedalus provides only a few primitive parsers:

- the *empty string* parser `^v`, which consumes no input and succeeds with semantic value `v`,
- the *empty language* parser **Fail** `msg`, which never succeeds and produces an error message,
- the *singleton* parser `$[c]`, which extracts the first element of the input, as long as it belongs to character class `c`, and
- the *string* parser **Match** `"string"`, which matches a string literal; semantically, this is equivalent to sequencing multiple singleton parsers and storing their results in an array.

Note that our singleton parsers are a generalization of the terminals found in traditional grammars. The character classes serve to provide a more compact notation for large numbers of alternatives. For example `$['a' .. 'z']` is more compact than composing 26 terminal parsers in parallel.

3.2 Sequential Composition

The sequential composition of parsers `P1` through `Pn` is written as `{ P1; ...; Pn }`. This succeeds if each sub-parser succeeds in turn on the input. By default, the result is the result of the last parser, but that may be changed by explicitly defining the special variable `$$`. The results of intermediate parsers may be named with the **let** keyword. If a result is named, but the **let** keyword is omitted, then the result of the parser is a *record* that contains the corresponding field (more on this in Section 5). Finally, Daedalus also supports a less noisy *layout* based notation for sequencing, using the **block** keyword. The examples on the right illustrate sequencing parsers.

```
def Add2 =
  block
    let x = BEUInt64
    let y = BEUInt64
    ^ x + y
```

```
def StatementSemi =
  block
    $$ = Statement
    $[';']
```

3.3 Parallel Composition

Parallel composition allows multiple parsers to examine the same part of the input. Daedalus supports two flavors of parallel composition:

- *biased* composition `P1 <| P2` (or **First** in its layout form) parses the input with `P1` and will fall back to `P2` if, and only if, `P1` fails; one may think of `P2` as an exception handler for `P1`.
- *unbiased* composition `P1 | P2` (or **Choose** in its layout form) parses the input with *both* `P1` and `P2`, and may result in multiple results if the grammar is ambiguous.

The parallel composition operators require their sub-parsers to produce the same type of semantic value. Parsers that construct different types of semantic values may be composed in parallel by injecting the values into a common *tagged union* type. Section 5 has more details on tagged unions.

3.4 Repetition

The most primitive way to handle repetition in a parser is with *recursion*. Direct use of recursion, however, can be error prone and difficult to understand or analyze. Thus, Daedalus provides structured repetition constructs that can easily express common repetition patterns.

Kleene Star. A well-known construct from regular expressions is the Kleene-star, which sequences a parser `P` with itself some number of times. In Daedalus this is written as **Many** `P`, which applies `P` as many times as possible and returns an array of the results. **Many** also has a constrained form that bounds the number of iterations. For example, **Many** `(1..) P` will succeed only if `P` is applied at least once. Daedalus's data dependency support means the constraints need not be constants, and may be parse results. The "lazy" variant of this construct is **Many?**, discussed further in Section 7.

Kleene Star with State. A related iteration pattern is **many** `(x = s) P`, which—like **Many**—uses `P` sequentially as many times as possible. The difference is that in this form, `P` may depend on a piece of state `x`, which starts off as `s`. On each successful iteration, `P` computes a new value for `x`. The result of the whole **many** construct is the final value of `x`. On the right is an example that shows how to use **many** to parse a base 10 number.

```
def Number =
  many (value = 0)
    block
      let d = Digit
      ^ 10 * value + d
```

Iterating Over Collections. Daedalus provides two constructs for traversing collections, **map** and **for**. Both constructs succeed only if the body `P` succeeds for every element of the collection. They bind the variables `k` and `v` to successive keys and values respectively. (The key may be omitted, if it is not needed.)

```
map (k,v in c) P
for (x = s; k,v in c) P
```

The result of **map** is a collection with the same keys as `c`, but with values computed by the parser `P`. The **for** loop carries a piece of state `x`, which initially has the value `s`. On each iteration, `P` computes a new value for `x`. The result of **for** is the final value of `x`.

Relation Between the Iteration Constructs. The four iteration constructs of Daedalus are closely related and are the result of choices in two orthogonal design dimensions:

(1) *How does iteration terminate?*
   - **Many** and **many** never fail; body failure terminates iteration.
   - **map** and **for** propagate body failure; the argument container determines termination.

(2) *What is the result of iteration?*
   - **Many** and **map** collect values parsed by the body.
   - **many** and **for** update the iteration state with each body result and ultimately return it.

3.5 Branching

The most direct way to introduce data-dependent behavior in Daedalus is to make decisions based on a parsed value using **case-of** (or **if-then-else** for boolean values). The **case-of** construct chooses the first alternative whose pattern matches the provided value. If none of the alternatives match, it behaves like **Fail**. Note that while **case** resembles parallel composition, there is also an important difference—in particular, **case** makes its decision based on the value being examined, and will only ever execute one of the branches. If the branch fails, then the entire **case** construct fails. This is in contrast to biased choice, which will consider further alternatives if an earlier one fails, and also different from unbiased choice, which considers all alternatives at once.

3.6 Coercions

Daedalus is a strongly typed language, but it also provides mechanisms for coercing between values of different types. The first one, `v` **as** `T`, is a *safe static* coercion of the value `v` to type `T`. The coercion is safe in the sense that no precision is lost when coercing the value to its target type. The safety of the coercion is determined statically, based on the type of the value and the target type. For example, it is always safe to coerce from **uint** 8 to **uint** 64.

On occasion, it is also necessary to perform a *static lossy* coercion, which is written `v` **as!** `T`. This allows to convert a value while possibly losing some precision. For example, converting a **uint** 64 to a **uint** 8 this way would retain only the 8 least significant bits of the original value.

Finally, Daedalus also supports a *dynamic safe* coercion, written `v` **as?** `T`, which will perform a check at runtime to determine if the value `v` may be coerced to type `T` without losing any information. For example, we may coerce from **uint** 64 to **uint** 8, but this will succeed only if the input value is less than 256. Otherwise the coercion behaves like the failing parser **Fail**.

4 EXPRESSION NOTATION FOR SEQUENCE PARSERS

Daedalus supports an *applicative* notation for sequences of parsers which often helps to write more concise and readable parsers. The notation was inspired by the bracket notation for applicative programming with effects [18], adjusted as described in this section. The central idea is that parsers can be used in contexts that expect a semantic value, as long as the whole construct is in a parser context. Consider, for example, `f P Q`, where the semantic function `f` is applied to two parsers `P` and `Q`. This is desugared into `{let x = P; let y = Q; ^ f x y}`, which parses `P` first, then `Q`, and finally applies `f`. Similarly, one can examine the result of a parser with a **case** expression. Using a value in a context expecting a parser is a special case of the function call rule, corresponding to a zero-arity function application. For this reason, most of the time specifications need not use the `^v` parser (consume nothing and return a value) directly, and write just `v` instead.

The applicative notation does not require special bracketing and works everywhere in a specification. Instead of bracketing, we rely on context to determine when to desugar applications. Since Daedalus supports type inference we need a mechanism to distinguish semantic values and parsers *syntactically*. To do so, it is sufficient to be able to distinguish between parser names and semantic value names, which we achieve by partitioning the set of identifiers:

- semantic values use names starting with a *lower-case* letter,
- parsers use names starting with an *upper-case* letter,
- character classes use names starting with `$`.

The restriction on character class names also allows us to promote character class names to parsers, which allows us to just write `$x` instead of `$[$x]`.

5 SEMANTIC VALUE TYPES

Daedalus supports a rich language of semantic values, including booleans, various numeric types, optional values, arrays, association-maps, user-defined bitdata [38], and algebraic datatypes (ADTs). Most of these are standard and available in most programming languages. In this section we focus on two of the less common type phenomena in Daedalus: *bitdata declarations* and *automatic synthesis of algebraic datatypes*. These both combine the declaration of a type and its representation in a single declaration, making specifications more concise and easier to understand and maintain.

5.1 Bitdata

Bitdata declarations introduce types with explicit bit-level representation. They are particularly suited for working with packed bit-fields, and values whose representation needs to set certain bits. In particular, *enumeration types* are a special case of bitdata.

```
bitdata Packed where
  x: uint 8
  0xFFFF
  y: uint 8
```

```
bitdata Uni where
  value = { get: Packed }
  null  = 0
```

Some bitdata declarations appear on the right. On top is a bitdata *record*, defining a new type called `Packed` to be represented with a 32-bit word. The record has two fields, `x` and `y`, which occupy the most and least significant 8 bits of the word, respectively. The middle 16 bits of the word are always 1.

On the bottom is a bitdata *union*, defining a new type called `Uni`, which is also represented with a 32-bit word. Values of this type may be created with two constructors, `null`, represented as `0`, and `value` which contains a valid representation of a `Packed` value. Values may be inspected using pattern-matching as usual, except that instead of generating additional "tag" bits, Daedalus will compute a discriminator based on the representations of the constructors.

The Daedalus compiler uses these declarations to synthesize code, such as field accessors, constructors, pattern recognizers, and value validators, which greatly simplifies the—usually rather error prone—process of working with such values.

While values of bitdata types may be constructed with constructors, just like any other algebraic datatype, it is also possible to coerce the underlying representation type into a value of the bitdata type (e.g. convert a **uint** 32 to a `Packed` value). Daedalus analyses the bitdata declaration to find which bit patterns correspond to valid values. If all bit patterns are valid, no work is done at runtime. However, if there are values in the representation that correspond to no value in the bitdata type, then the coercion must be dynamic. For example, a coercion to `Packed` needs to check that the middle 16 bits are all 1. The analysis uses BDDs to represent sets of possible bit-patterns [25].

Indeed, it is quite common to parse a bitdata value in two steps: (1) parse the representation, and (2) coerce the representation to the bitdata type. This allows separating byte order concerns from validation of the value produced. Thus, to parse a big-endian `Packed` value, we would first use the 32-bit big-endian parser, like this: `BEUint32` **as?** `Packed`.

5.2 Automatic ADT Synthesis

In traditional languages with algebraic datatypes, datatypes must be declared before values of that type. This is useful because the declaration of the type serves as the contract between various functions that use the given type. Daedalus supports this mode of operation; however, we also provide a shortcut for a very common case. In particular, since Daedalus is a parsing DSL, most of the time we are writing parsers for various datatypes, while complex computation over the resulting types is relatively uncommon. As such, it is quite convenient to *infer* the declaration of an algebraic datatype from its parser. Consider, for example, the header for a typical binary data format. One might declare a record with all the fields found in the header, and then write a parser

that produces a value of this type. Alas, the resulting specification typically contains quite a lot of duplication. To help with this Daedalus does not require explicit declaration of parser result types.

In our design, a parser declaration defines *both* a parsing function and a type. The type introduced by a parser refers to the type of semantic values it produces. For example, the code on the right declares a parser called `Point`, which implicitly defines a new record type, also called `Point`, with two **uint** 8 fields, `x` and `y`. It also defines a parser called `PointX` which parses points in a different way. We use an explicit type annotation to specify that `PointX` constructs a value of type `Point`, instead of introducing a new type. The name `PointX` becomes a *type alias* for `Point`.

```
def Point =
  block
    x = UInt8
    y = UInt8
```

```
def PointX : Point =
  block
    x = UInt8
    y = 0
```

We implement this feature by using *overloaded constructors* for values of algebraic types. When constructing an ADT value, we do not pick a specific type, but instead keep track of constraints on what the type needs to support. (In the case of a record type, the constraints might specify that the concrete type should have certain fields with certain types.) We then do type inference as usual, which may resolve the concrete type of the constructor. After validating a declaration, we check if any constructor types are still unresolved; if so, we use the collected constraints to generate a minimal satisfying new type, and resolve the types to refer to the newly generated type.

Tagged Unions. The same system also works for parsers constructing tagged union values, which may also be recursive. A tagged union constructor, `{| C = v |}`, produces a constraint relating the type of constructor `C` to the type of its field, in this example `v`. If inference does not resolve the resulting type, we generate a new tagged union type containing only the constructors used in the definition. Since working with tagged unions is common, we also provide tagged variants of the **First** and **Choose** constructs, which are syntactic sugar for composing the untagged version with the appropriate constructor. The code on the right illustrates how one might define a parser that behaves like **Many**, but instead of producing an array, constructs a linked list of the parsed values.

```
def List P =
  First
    Cons = Node P
    Nil  = Accept
```

```
def Node P =
  block
    head = P
    tail = List P
```

6 LOOKAHEAD AND STREAM MANIPULATION

While text-based formats are often processed in a linear fashion, processing some binary formats requires non-sequential access to the data that is being parsed. For example, many formats have a table of contents that contains offsets to various other elements. Rather than processing everything linearly, an application may read only the table of contents, and then continue at a particular sub-component at some other part of the input. Daedalus supports this, and other non-linear parsing use cases, by allowing users to manipulate the data being parsed. At the lowest level, this is done with four primitive parsers that handle abstract semantic values of type **Stream**:

- **GetStream**, which never fails, and returns the current stream;
- **Drop** `n` `s`, which computes a new stream corresponding to `s` advanced by `n` bytes, and fails if `s` does not contain enough data[1];
- **Take** `n` `s`, which computes a new stream containing the first `n` bytes of `s`, and fails if `s` does not contain enough data;
- **SetStream** `s`, which never fails and modifies the current data being parsed—parsers that are sequentially composed with this construct will process `s` instead of the current stream.

[1]While **Drop** may be defined using **Many**, we made it a primitive to keep stream manipulation self-contained. Also, it is quite convenient—for both users and the implementation—to make advancing the stream explicit.

Using these constructs we can express a wide variety of non-linear parsers, including complicated use cases, such as documents that are partially compressed (or encrypted), so parts of the input need to be unpacked before they can be parsed. The simplified example given to the right shows parsing a table of object offsets, jumping to each offset, and parsing the individual objects. The result is a sequence of semantic values resulting from parsing the objects.

```
block
  let base = GetStream
  Many block
    let offset = Word64
    let here = GetStream
    SetStream (Drop offset base)
    $$ = ParseObject
    SetStream here
```

These basic constructs are quite low-level, and using them can lead to specifications that are hard to follow and error prone. Therefore, we provide a collection of derived combinators that capture various common patterns. Examples include `Chunk n P`, which parses the following `n` bytes using parser `P`, and `LookAhead P`, which applies parser `P` to the current input without consuming it.

Semantics. **SetStream** assigns the current input stream. One might argue that this is too powerful. The Daedalus standard library also includes a more restricted variant, **WithStream** `s P`, which applies parser `P` to input `s` but, crucially, does not modify the current stream. This is sufficient for many use cases. Interestingly, while using **SetStream** pretty much forces the semantic interpretation of a parser to be a *stream transformer*, if we restrict to **WithStream**, it is still possible to interpret parsers as sets of strings, which is much closer to the traditional interpretation of grammars. (Since Daedalus is data dependent, the sets do not contain just strings, but pairs of strings and semantic values, and since the combination of **GetStream** and **WithStream** still supports look-ahead, instead of using strings, we need to use "strings in context", sometimes referred to as *cursors* in the input, but the general interpretation of a parser can still be just a set, rather than a function.)

7 EAGER VS. LAZY

Along with biased and unbiased choice for parallel composition, Daedalus also has two forms, "eager" and "lazy", for the looping operators where termination is determined by the success of the parser (i.e, unconstrained **Many** and **many**). Eager iteration consumes as much input as possible, and is closely related to biased choice: we choose to consume input whenever possible rather than terminating the loop. Lazy iteration is correspondingly related to unbiased choice: conceptually, we explore both the alternatives of terminating the loop and of doing one more iteration.

The two forms provide different trade-offs. The biased/eager constructs, popularized by parsing expression grammars [10], have the benefit that only one of the alternatives will be selected, and as a result, they cannot introduce ambiguity. They are also relatively simple to compile efficiently, especially to recursive descent code, because the path to take depends only on the specific parsers being composed in parallel. They can, however, be somewhat error prone. One might avoid writing an ambiguous grammar, and instead write the *wrong* grammar. One might compose alternatives in the wrong order, or not notice that the input accepted by one parser in a loop may overlap with the input accepted by the next parser after the loop. The unbiased/lazy operators avoid these ordering pitfalls, but do introduce the possibility of ambiguity. In general, generating efficient parsers for grammars using the unbiased/lazy constructs is also harder, unless the determinization analysis described in Section 9.1 can resolve the choice based on limited look ahead.

8 REVISITING PDF

This section presents a fragment of our PDF specification, as an extended example. As we noted in Section 2, the data of a PDF file is stored as a collection of object declarations, and Figure 2 describes their structure in Daedalus.

The left-hand side of the figure shows the high-level structure of an object: it has a name (two numbers), and a definition enclosed between `obj` and `endobj`. The definition is either a `Value` (a normal COS object), or a *stream object*, which is how most actual data in a PDF is stored. Streams contain things like the pages of the PDF, images, and even other PDF objects.

The right-hand side of the figure shows how to process streams: these contain a header, which is a COS dictionary, and some data enclosed between `stream` and `endstream`. The `Chunk` combinator from our standard library restricts the input to the length of the stream data. While streams are terminated with `endstream`, this does not determine the amount of data in the body. Instead, the length of the data appears in the header. As an additional complication, the length in the header may be a *reference* to another object that contains the actual number[2]. We encapsulate this pattern in `LookupResolve`. This operation gets a value from a dictionary, but if it sees a reference, it also resolves it by parsing another object at an entirely different position in the file. To resolve the reference we need access to the `xref` table, which we access via a piece of user-defined parser state—we could have parameterized all declarations by the `xref` table, but this was quite messy. The final complication is that the header may specify transformations to be applied to the bytes (e.g., decompress them, or apply various encodings). This is abstracted in the function `ApplyFilters`, which processes the header and applies transformations to the current input. In our implementation the transformations are implemented as external primitives, which often use existing libraries.

```
def TopDecl = block
  ManyWS
  id   = Token Natural
  gen  = Token Natural
  KW "obj"
  obj = TopDeclDef
  Match "endobj"

def TopDeclDef = block
  let val = Value
  First
    stream = Stream val
    value  = val
```

```
def Stream (val : Value) = block
  header = val is dict
  Match "stream"; SimpleEOL
  body = StreamBody header
  KW "endstream"

def StreamBody header = Token
  (Chunk (StreamLen header)
         (ApplyFilters header GetStream))

def StreamLen header = block
  let lenV = LookupResolve "Length" header
  NumberAsNat (lenV is number) as? uint 64
```

Fig. 2. Daedalus specification for parsing PDF objects.

9 IMPLEMENTATION

We implemented a few different tools for working with Daedalus specifications, in particular:

- `daedalus`, for interpreting specifications, visualizing parser results, and compilation to either C++ or Haskell;
- `talos`, which may be used to synthesize inputs accepted by a grammar; and
- `daedalus-language-server`, providing LSP support [51] for editor integration.

The tools rely on a collection of inter-dependent libraries for processing and analyzing Daedalus specifications, and in this section we describe some of the more notable aspects of our implementation. The overall structure is not unlike that of traditional compilers for general purpose programming languages, and processing a specification happens in a sequence of passes which may be grouped in three components, each with its own intermediate representation (IR):

[2]PDF allows this so that generators can write the length as a separate object, after generating the body of the stream.

(1) *The Front-end.* The front-end passes mostly deal with validating specifications, including parsing, name resolution, and type inference.
(2) *The Core.* Passes in this group use a simplified IR, containing only the essence of Daedalus. The code in the core mostly deals with analysis and transformation of specifications.
(3) *The Back-end.* Passes in this group use a low-level CFG-like IR. The back-end code mostly deals with code generation, and analyses related to code generation.

The different tools stop at different parts in the pipeline. For example, `daedalus-language-server` uses only the front-end, `talos` works mostly with the core representation, and `daedalus` exercises the full pipeline. We also have two important passes that translate *between* IRs, described next.

Specialization. This pass translates from the front-end to the core IR. The biggest difference between the IRs is that the front-end supports polymorphism and productions that are parameterized by other productions, while in the core IR all declarations are monomorphic and may be parameterized only by semantic values. Besides desugaring various constructs, the specialization pass performs a whole-program analysis, starting at the entry points of the specification, and generates specialized versions for each use of a polymorphic function, or a production parameterized by another production.

Compilation. This pass translates from the core IR, to the back-end CFG. The main purpose of this pass is to make explicit aspects that are left implicit in the core. In particular, here is where we make the parsing algorithm we use explicit. Our implementation compiles Daedalus using recursive descent on the structure of the grammar, but this is an implementation choice and in principle one could consider other implementation strategies. At this point the input to the parser becomes an explicit parameter to each declaration. Control flow also becomes explicit (biased choice is explicitly implemented with backtracking, we choose various calling conventions for calling functions, etc.).

Validation. We treat each intermediate representation as a small language in its own right. As such, each intermediate representation has a validator, checking various invariants of the IR (e.g., that the IR is type-correct), as well as a precise semantics, in the form of an interpreter. This serves a dual purpose. It provides documentation, capturing explicitly many aspects of the code that might be left implicit otherwise, and it makes it possible to *test* the various passes of our implementation, to ensure that each pass preserves both the well-formedness of the IR and also its semantics.

9.1 Grammar Analysis and Transformations

In this section we describe a few of the grammar-level transformations we perform—these preserve the semantics of the specification but are intended to improve the generated parsers, or make the specification more suitable for analysis. A lot of the transformations are standard for a programming language, although perhaps not as common in a parser generator. Examples include constant folding, definition inlining, and case inlining (sometimes referred to as case-of-case optimization).

This highlights one of the trade-offs between a standalone and an embedded DSL: standalone DSLs, like Daedalus, need to do this work explicitly, while embedded DSLs (e.g., a parser combinator library in Haskell) would get these for free from the compiler for the host language. On the other hand, we can also do some transformations that are specific to the semantics of Daedalus, that might be hard to spot for a general purpose compiler. We outline these next.

Unused Semantic Values. It is common for specifications to examine the underlying input, but not make use of the resulting semantic value. Consider, for example, skipping some white space using a parser like **Many** `$white_space`. The resulting semantic value is an array of bytes, but it is unlikely that this would be used for anything, and it would be wasteful to create it, only to

discard it. Embedded parser combinator libraries typically leave it to the writer of the specification to avoid this (e.g., by using a different combinator that behave likes **Many** but throws away the result). In `daedalus` we generate custom match-only variants for parsers, and then we transform specifications to use the match version in contexts where the semantic value is ignored.

Determinization. This transformation is fairly standard for parser generators, and is similar to the algorithm for converting an NFA into a DFA following derivative based techniques [12]. In our language-based settings, this amounts to replacing parsers composed in parallel with a **case** parser, when we can find a prefix of the input that can discriminate between the alternatives. The code on the right shows an example of this transformation. The transformation performs algebraic rewriting using derivatives and zipper, inspired by Edelmann et al. [28], extended to several bytes lookahead.

Original code:

```
Choose
  { $['a']; P }
  { $['b']; Q }
```

Transformed code:

```
case $any of
  'a' -> P
  'b' -> Q
```

Factor Out Non-failing Parsers. We also found it useful to reduce the scope of biased choice, by factoring out non-failing parsers. The transformation is illustrated on the right. `Q` is a parser that cannot fail. `Optional P` is a parser that behaves like `P`, but instead of failing returns `nothing`.

We implemented a static analysis to annotate non-failing parsers, which is needed to ensure the validity of the transformation. When combined with case-inlining this rule helps preserve the linearity of state in state transforming parsers, such as the body of the **many** loop. Recall that this loop executes the body, and on success produces a new state value and does another iteration. On failure, we terminate using the old state value. Factoring the creation of the new state out of the parallel composition reveals that we need not roll back to the old state, which in turn allows updating it in place [59].

Original code:

```
{ let x = P
; Q } <| R
```

Transformed code:

```
case Optional P of
  nothing -> R
  just x  -> Q
```

9.2 Interfacing With an Application

While one may write a standalone format validator entirely in Daedalus, far more commonly the generated parser needs to be integrated with an existing external application. The most basic use case is to generate parsing functions for a specification's entry points. In this section, we outline some additional features that we found to be useful for applications using Daedalus parsers.

Calling External Code. Daedalus provides a rich language for manipulating semantic values but, in practice, it is often useful to be able to reuse existing code. For example, while processing a PDF document, we may need to decrypt part of the input before parsing it, which is best done by calling existing cryptography code. Daedalus supports both externally defined semantic functions, and externally defined parsers, the main difference being that parsers can access the underlying input and can fail, while semantic functions cannot. Of course, one must be careful when using external functions, as the generated parser would inherit any problems they have.

User-defined State. Another useful feature is the ability to pass some application specific data to external calls. This data is provided by the application when calling a parser, and is not visible to the specification. However, when the parser calls an external function, it makes this data available to the external code, which can manipulate it as needed. We used this feature in our PDF parser to carry around a cache of PDF objects without needing to thread it through the entire specification. Whenever the parser needs to resolve a reference it calls an external function that has access to the cache and wraps the object parser; this means we never parse any object more than once.

Parsing Partial Data. In some scenarios, a parser may have to process partial inputs (i.e., the entire input is not available when parsing begins). For example, applications that parse data arriving from a network do not get to choose when the data arrives or in what chunks. In some cases explicit external calls to get more data may work, but even when workable this is intrusive. To handle these cases, `daedalus` supports a compilation mode that uses a custom *lazy stream* datatype to represent the input. (Normally the input is an array of bytes.) This datatype supports being extended with incremental chunks of data, and distinguishes between *partial inputs* (i.e., cases where we are out of data, but more data may be coming), and *finished inputs*, where we know that no more data is available. This is important because Daedalus supports look-ahead and matching on the end of input. When a parser tries to access data at the end of a partial stream, the parser is suspended, and control is transferred to a callback that will either add more data to the stream or mark it finished. This makes it possible to write applications such as a single threaded web-server with one parser per connection. Whenever data arrives over a connection, the corresponding parser is resumed.

9.3 The C++ Back-End

In this section we give a brief overview of the code generated by our C++ back-end. For each entry point to the parser we generate a parsing function, as shown on the right. On error, there are no results, and the error object contains specifics. Otherwise there will be one result, or more if the grammar is ambiguous. The run-time system provides classes for semantic values of built-in type, such as numbers and arrays. User-defined types are represented with custom classes generated by the compiler. Figure 3 shows the class generated to represent a sum type. Broadly, these classes provide ways to initialize a value, examine it, and do memory management. We also generate support functions such as comparison and printing.

```
void parseMain(
  DDL::ParseError<DDL::Input> &error,
  std::vector<DDL::UInt<8>> &results,
  DDL::Input a1);
```

Memory Management. The generated code uses reference counting to manage allocated memory, with an additional ownership discipline that distinguishes between *owned* and *borrowed* uses. An owned use of a value is reflected in the reference count, while borrowed ones are not. Borrowed uses therefore require care. (In particular, the compiler never stores borrowed values in other data structures.) It is always possible to convert a borrowed value into an owned one by incrementing its reference count. Doing these manipulations manually is quite error prone, so the `daedalus` standard library supports the standard C++ hooks for adjusting the count when values are copied or destroyed. The compiler-generated code includes all reference-count operations. However, an application consuming the result of the parser needs to be aware of the memory model.

Memory Management Validation in the Compiler. To ensure that we generate correct memory management code, we perform a validation pass after we insert the reference count increment and decrement instructions. At this point in the pipeline, the code is a control-flow graph in SSA form, where each basic block takes all of its parameters explicitly, and the parameters are annotated with ownership information. The validation pass tracks if a variable is borrowed or owned, and owned parameters are assumed to start with a reference count of 1. We simulate each instruction in the block, keeping track of the reference counts, and check the following conditions:

- we never use an owned variable that has a count of 0,
- we never use a borrowed variable when an owned one was required,
- we never decrement the reference count a borrowed variable,
- at the end of the block all owned variables are at reference count 0.

```
namespace PdfCos {
  class Value : public DDL::IsBoxed {
    DDL::Boxed<PdfCos::Private::Value> ptr;
  public:
    /* Constructors */
    void init__bool(DDL::Bool);
    void init_dict(DDL::Map<DDL::Array<DDL::UInt<8>>, PdfCos::Value>);
    ...
    /* Selectors */
    DDL::Tag::Value getTag();
    DDL::Bool get__bool();
    DDL::Map<DDL::Array<DDL::UInt<8>>, PdfCos::Value> borrow_dict();
    DDL::Map<DDL::Array<DDL::UInt<8>>, PdfCos::Value> get_dict();
    ...
    /* Variant dispatch */
    using Pat__bool = DDL::Pat<DDL::Tag::Value, DDL::Tag::Value::_bool>;
    using Pat_dict = DDL::Pat<DDL::Tag::Value, DDL::Tag::Value::dict>;
    ...

    template<typename Cases>
    auto sum_switch(Cases&& cases);

    /* Memory Management */
    void copy(); /* increment refcount */
    void free(); /* decrement refcount */
  };
}
```

Fig. 3. C++ class for the PDF COS object type, which is a sum with a number of alternatives, including `_bool` and `dict`. To save on space we only show the pieces for these alternatives; the others follow a similar pattern. The `DDL` namespace includes the run-time system's semantic value types, such as `DDL::Bool`.

10 EVALUATION

There are three competing concerns in parser development, and indeed in software engineering in general: ease of implementation, performance, and safety. We evaluate Daedalus on a suite of microbenchmarks in comparison with five popular parser generators or combinator libraries, measuring execution speed to gauge performance and using the number of lines of code as an imperfect proxy for ease of implementation.

Daedalus was also subject to two evaluations conducted by third parties. One measured the performance of and defects encountered in four PDF parsers using tens of billions of PDF documents. The other measured defects using popular static analysis tools. We report those results here.

10.1 Microbenchmarks

To evaluate implementation effort and performance, we wrote parsers for a few different formats in Daedalus and also using other tools. We are not aware of a parser generator that supports all the formats in question, so instead we just picked the tool for the relevant language (C++ or Haskell) that seemed most suitable for the job. The performance benchmarks were done with the `hyperfine` [22] tool on one of our development machines[3]. Table 1 summarizes the results. The

[3]Ubuntu 23.04 OS, 12th Gen Intel® Core™ i9-12900H CPU, 32GB memory, M.2 2280, Gen 4 PCIe x4 NVMe SSD

Table 1. Benchmarks. Times are in seconds, averaged over ten runs. Standard deviations are < 5%, except for the 0.492 time, whose standard deviation was 9.5%. In parens are approximate MB/s.

	C++			Haskell			
	Daedalus	Kaitai	ANTLR	Daedalus	attoparsec	bson	happy
LEB128	0.313 (137)	1.609 (27)	-	0.492 (87)	1.459 (29)	-	-
BSON	1.508 (67)	3.953 (26)	-	1.564 (65)	-	1.593 (63)	-
S-Expr	4.055 (25)	-	9.570 (11)	5.781 (17)	-	-	6.834 (15)

rest of this section provides more details on the benchmarks, and while these are by no means exhaustive, we hope that the results in Table 1 demonstrate that our tools can generate performant parsers that compare quite favorably with existing alternatives.

LEB128. This is a little-endian variable sized encoding of numbers in base 128, which is used as a component of many binary formats. The format is pretty simple, but requires some bit manipulation to determine how many bytes to process. The sample task was to parse and add up 10^6 randomly generated numbers of length between 1 and 8 bytes, resulting in an input of size 43MB. We compared the Daedalus parsers against a C++ parser generated with the Kaitai Struct compiler (version 0.10), and against a Haskell parser written with the `attoparsec` [11] library (version 0.14.4).

BSON. This is a binary rendering of JSON, used by MongoDB. We chose this format because it has many features that are common in other binary formats, for example: objects of limited size, various encodings of strings, and nested formats. (The strings are themselves encoded in UTF-8, which we need to validate.) The sample task was to parse a randomly generated BSON document (100MB) and add up the length (in bytes) of all strings. The BSON documents contain strings, numbers, and nested documents. We compared our parsers to a C++ parser generated with the Kaitai Struct compiler [43] (version 0.10), and against the Haskell `bson` library [64] (version 0.4.0.1).

S-expressions. This is a tree encoding used by a number of formal method tools (e.g., SMT-LIB, Racket). We chose this format because it requires matching of parentheses, and is suited to more traditional text-based parsing tools. The sample task was to parse a large randomly generated S-expression (100MB) and count the number of atoms. We compared the Daedalus parsers against a Haskell parser written with an `alex` [15] (version 3.4) generated lexer and a `happy` [5] (version 1.20) generated parser, and also against a C++ parser generated with `ANTLR` [63] (version 4.7.2).

10.2 Comparing the Specifications

Table 2 shows the lines of code for each microbenchmark implementation, excluding standard libraries and generated code. While the total lines of code do not vary dramatically between these examples, we find there is a qualitative difference in the programming experience. In the remainder of this section, we provide a brief discussion of the more notable stylistic differences between the various ways to write the parser specifications.

Basic Syntactic Structure. The specifications of the parsers we benchmarked may be grouped in three categories, based on their notational philosophies:

- Custom grammar notation (`ANTLR`, `happy`, `alex`)
- Embedded in Haskell (`attoparsec`, `bson`)
- Based on YAML (`Kaitai`)

There are various trade-offs between these: for example, using a highly structured notation, such as YAML, makes it easy to explain the basic syntax of the language, and write tools that parse

Table 2. Lines of code in benchmark implementations, excluding standard libraries and generated code. We omit bson because it is a hand-tuned implementation in Haskell and does not have meaningful non-library code to measure.

	Daedalus	Kaitai	ANTLR	attoparsec	happy
LEB128	28	37	-	50	-
BSON	112	202	-	-	-
S-Expr	15	-	11	-	52

specifications. On the other hand, subjectively, these specifications are a bit harder to read for a human, as they tend to be more verbose, and contain a fair bit of repetition. Daedalus is somewhere between the first two options: our specifications are structured in a way that is similar to a programming language like Haskell, but we also provide some custom notation to cover common cases (e.g., ways to write character classes, constraint repetitions combinators, etc.).

Organizing The Specification. A Kaitai specification of a parser roughly contains two sections: one specifies the layout of the data to be parsed, while the other specifies various attributes that may be computed from the matched fields. This separation between matching and computation is also present in the non-dependent parsing tools (`ANTLR`, `happy`). The three differ in how the computation is specified. Kaitai Struct uses a custom computation language, `happy` uses Haskell, and `ANTLR` only specifies a name, which is linked to computation in a backend-specific way (e.g., it can be mapped to a visitor class).

In contrast, Daedalus and monadic parser combinator libraries allow computation and matching to be intermixed. This has both positive and negative aspects—by mixing computation and matching sometimes we can generate more efficient parsers, because we can avoid generating intermediate structures. Some intermixing like this is unavoidable in data-dependent grammars. However, too much of it may lead to specifications that are hard to understand, and generally, we find that separating matching and computation leads to specifications that are easier to understand.

Like Kaitai Struct, Daedalus also uses a custom computation language; however, it is quite a bit more expressive and has, subjectively, more intuitive semantics. The restrictions of the Kaitai Struct language showed up in the LEB128 parser, where one needs to combine a sequence of 7-bit numbers into a single number. In Daedalus we do this with a loop. By contrast, the Kaitai computation language does not support iteration, so the loop has to be manually unrolled. This is undesirable, because it is error prone, and leads to duplication which makes the specification harder to read. Another limitation we encountered is that unless explicitly specified, computation in Kaitai is always done at a fixed implicit numeric type, which may or may not be what is needed in a particular case. This seems very error prone, as we encountered when we tried to use the specification from the Kaitai Struct website and got an incorrect parser. In contrast Daedalus uses type inference and strong typing to significantly reduce such errors.

10.3 Red Team Evaluation for PDF

The Daedalus PDF parser was examined for vulnerabilities and defects by a commercial third-party security auditor. Three other PDF parsers were also evaluated in the same audit: XPDF [69], Parsley [52], and a parser developed by BAE Systems. Each parser was evaluated continuously for 7 days using tens of billions of inputs drawn from PDF corpora scraped from open government documents, Common Crawl [17], and generated files constructed to contain syntactic and semantic malformations. Errors and anomalies were investigated, and each tool was scored by the number of unique defects detected. Table 3 presents these results. The errors in the 5 defect files for our

Table 3. Evaluation results for PDF parsers reported by third-party auditor. Five defects were discovered in the hand-written C++ wrapper driving the Daedalus parser. No defects were discovered in the Daedalus parser itself.

	Unique Defects	*K Lines of Code*	*Tests (Billions)*
BAE PDF	34	71	36.8
Daedalus PDF	5 (0)	4	72.8
Parsley PDF	7	14	41.6
XPDF	41	80	10.0

parser were in two categories:

- 4 of the defects were because our parser ran out of stack: the test files were hand-crafted to create an extremely nested parser context (e.g., many repeated `[` characters, which to the parser looks like a very nested array). Our mitigation was to change the grammar to impose limits on the nesting of values, which is not a general solution. Motivated by this problem, we considered various approaches to impose global limits on the resource consumption by the parser, but have not yet come up with a satisfactory solution.
- 1 of the defects was due to an error in the application that called the Daedalus generated parser: when the application determined the location of the `xref` table, it did not validate that the specified offset was within the limits of the buffer being parsed.

Size of implementation. Each implementation was measured by the number of lines of code in C++ header and source files. The Daedalus parser implementation comprised 2 KLOC lines of Daedalus specifications, plus 2 KLOC for the C++ wrapper, which is much smaller than the other tools. However, a direct comparison is difficult; XPDF, for example, includes business logic for displaying PDF files that is not possible to decouple from its parser. The Daedalus implementation does not include similar business logic.

Performance. Each tool was evaluated continuously for 7 days. The `Tests (Billions)` column in Table 3 shows the number of PDF documents each implementation was able to parse in that time. The Daedalus parser was able to process more than 70B documents, nearly twice that of the next-fastest tool, showing that its generated code is performant.

Safety. The Daedalus implementation had the fewest defects. Digging deeper, it turns out these defects arose in the hand-written C++ wrapper that calls into the Daedalus-generated parser—recall from Section 2 that the Daedalus PDF implementation comprises a C++ parsing library generated from the Daedalus PDF specification along with a C++ wrapper that dispatches calls to the parser. No defects were detected in the parser, and the defects in the wrapper have since been fixed.

10.4 Evaluation with Static Analyses

A second third-party evaluation [4] shows similar results to the red team exercise; they applied static analysis tools [13, 48, 68] to three parser DSLs: DFDL [50], Kaitai Struct [43], and Daedalus, using a representative grammar included with each tool to generate C++ parser implementations. Table 4 presents the results.

Table 4. Static analysis for PDF parsers.

	Weaknesses
Apache Daffodil	20
Kaitai Struct	22
Daedalus	1

This evaluation discovered one warning in the Daedalus implementation, where a call to open did not prevent opening a symlinked file. In contrast, the evaluation discovered 42 weaknesses in

the code generated by Apache Daffodil and Kaitai Struct, including buffer overflow vulnerabilities and missing bounds checks.

11 ANALYSES TARGETING DAEDALUS

When deciding between an embedded or standalone DSL, amenability to analysis is a key consideration. Embedded parser DSLs like Parsec [21] (Haskell) and Hammer [35] (C/C++) are deeply coupled with their host languages, to the point where meaningful analyses of the resulting parsers require reasoning about the entirety of the host language as well as the DSL. Daedalus is a simpler language than Haskell and C/C++, and its implementation has a well-structured syntax and evaluation semantics—making it easier to write analyses targeting Daedalus-specified data formats. In this section, we briefly describe two such analyses: *document synthesis*, which automatically generates documents that satisfy a given Daedalus-specified parser, and *cavity detection*, which discovers locations in file formats that can enable code smuggling exploits.

11.1 Document Synthesis

Document synthesizers are remarkably useful when coupled with testing techniques like property-based testing [16] and fuzzing [8, 29]. These techniques involve testing a program by running it over and over with many different inputs. When testing a PDF processor, for example, a fuzzer attempts to generate hundreds of PDFs and ensure that none cause the program to crash. Of course, for such a fuzzer to be effective, it should mostly produce valid PDFs; invalid inputs will simply be rejected by the parser without reaching interesting program logic. To avoid this problem, developers often write ad hoc generators as part of their testing infrastructure that duplicate the parsing and validation logic of the program under test.

We built a document synthesizer for Daedalus-specified data formats, named Talos, which generates valid documents by analyzing a Daedalus parser. The Talos analyzer symbolically evaluates the Daedalus *core* syntax to collect constraints on parser input. From there, these constraints are passed to an SMT solver; a satisfying assignment to the constrained formula corresponds to valid input. To help improve the distribution of generated documents, Talos uses Boltzmann sampling [27] to sample the structure of a document—which branches to take, constructors to use, loop iterations to select, etc.—from a uniform distribution. This means that the solver need not make decisions about unconstrained choices, and the final distribution contains far more variety. To aid scalability, Talos leverages the structure of Daedalus semantics to decouple unrelated parts of the program via program slicing [67]. Constraints are then generated and discharged independently for each slice.

We have applied Talos to HTTP/1.1 and HTTP/2 Daedalus parsers and used the synthesized HTTP/1.1 messages to investigate the behavior of PicoHTTPParser, an open-source HTTP parser [55].

11.2 Polyglot Detection

A *polyglot file* can be parsed as more than one valid data format. A PDF/JPG polyglot may, for example, display a document when opened with a PDF reader while displaying an entirely unrelated image when opened as a JPG. Polyglots have been used to smuggle malicious code, such as disguising Java or PHP Archive (Phar) files as innocuous formats like JPG [36, 45].

Data formats are amenable (vulnerable) to polyglots when they admit *cavities*, contiguous bit ranges that are ignored by the parser. Formats like JPG and PNG are perfectly happy to ignore trailing data, admitting a cavity at the end of the document (suffix cavities). Other formats, notably PDF, will ignore data at the *start* of a document (prefix cavities) until finding a magic start token—for PDF, this is the string "`%PDF-version`". Together, JPG and PDF combine to form a simple polyglot: a document starting with JPG content (which PDF ignores) and ending with PDF content (which

Table 5. Feature comparison chart.

	Portable	*Binary*	*Data-Dependent*	*Noncontiguous*
yacc				
ASN.1	✓	✓		
Hammer		✓	✓	✓
Parsec		✓	✓	✓
nom		✓	✓	✓
Kaitai Struct	✓	✓	*limited support*	✓
Daedalus	✓	✓	✓	✓

JPG ignores). Albertini *et al.* identify and classify patterns of cavity locations in binary files and show how they combine to form polyglots [3].

To aid in detecting polyglots, we have built a context-sensitive, flow-insensitive dataflow analysis that identifies cavity locations in Daedalus specifications and determines whether they are prefix or suffix cavities. An instruction that reads from the input stream contributes to a cavity if (a) it is *unbounded*, that is, the data read does not flow to a control location that can lead to failure, and (b) it is part of a control flow cycle that does not contain other bounded reads. We have used this analysis to detect and reproduce Phar-based polyglots.

12 RELATED WORK

Daedalus is inspired by parser combinators [21, 35] and domain-specific data description languages, such as PADS [44, 70] and incorporates elements from each. But it also draws on additional features to meet the challenges inherent in tools for complex data formats. Table 5 lists our requirements: A tool should be portable, to define a data format once and generate parsers in multiple languages, and it should handle binary data with rich data dependencies and noncontiguous layouts.

Parser Generators. Some of the most lasting tools for defining parsers, motivated by a long history of work in the theory of formal languages [1, 37], are *parser generators*. Parser generators like yacc [61] and its successors [10, 63] are used to define a wide variety of context-free languages, including many of the programming languages that we use on a daily basis. However, many of these tools are not portable, as the target language is woven into the parser definitions, and they are less suited for binary formats, which often fall outside the set of context-free languages.

Interface Description Languages. When defining binary and textual formats, engineers often reach for *Interface Definition Languages* (IDLs). ASN.1 [6, 26, 46] is the paradigmatic example of an IDL; ASN.1 is used in hundreds of protocols and formats in domains such as network management, email, cellular telephony, air traffic control, and cryptography. Other IDL-based tools include DFDL (Data Format Description Language) [24] from the *Open Grid Forum*, protocol buffers [32], and Apache Thrift [60]. IDLs define an abstract notion of data value that are neither tied to a language representation nor an encoding method—thus ASN.1's name, *Abstract Syntax Notation*. Given an IDL specification, one can generate parsers (encoders) and printers (decoders) in various languages, providing significant portability and interoperability.

We have not considered printers in the design of Daedalus; this remains future work. But when it comes to parsers, Daedalus can express far more complexity than most IDLs. ASN.1 has local constraints (e.g., an integer is between two values) but no way to specify complex relational constraints between data fields. The same is true of parser generators: a yacc grammar could specify

some aspects of the PDF data layout, but significant re-parsing and validation would need to be done downstream to check the various well-formedness conditions inherent in the format.

Parser Combinators. The most flexible tools available for parsing complex formats are libraries or embedded DSLs. Many, for example Hammer [35] in C and Parsec [21] in Haskell, fall under the umbrella of *parser combinator* libraries; these provide a suite of "basic" parsers along with an interface for combining parsers together into more complex ones. There are even libraries like nom [54] in Rust and attoparsec [11] in Haskell that are designed to work with low-level formats like the ones this paper focuses on. Parser combinators are "shallowly" embedded in their respective programming languages, so they can do nearly arbitrary computations during parsing, applying complex rules and checks and ruling out malformed inputs without requiring further validation.

Another interesting point in the design space is the work on staged parser combinators [47, 53], which is half way between traditional parser combinators and parser generators: like traditional parser combinators, specifications are embedded in a programming language; however, staged evaluation is used to process the grammar at compile time. This eliminates interpretive overhead and allows generating efficient parsers.

Daedalus takes inspiration from the combinator approach to parsing, providing a compositional interface and flexible tools for validation, but it skirts the main downsides: language-dependence and analyzability. As a standalone parser language, Daedalus allows writing a single program to define a portable parser that can be compiled to many languages, preventing duplicated work and inconsistencies across implementations. We touched on the issue of analyzability in Section 11.

Binary Parser DSLs. IDLs like ASN.1 take a declarative approach to parsing binary formats but have limited ability to express noncontiguous formats with data dependencies and complex validation conditions. Binary parser DSLs like Kaitai Struct [43], Datascript [31], Everparse [57, 62], and Narcissus [23] take a similarly declarative approach but differ in several key ways: (1) primitives and combinators are fixed and limited to those applicable to binary formats; (2) rather than "computation in a host language," one has a small integer constraint language; (3) some method is provided to allow for non-linear layout of fields. Reducing the expressiveness of the parsing and computation languages allows extracting both the encoder and decoder. In contrast, Daedalus is Turing-complete, and its parser-combinator design provides strong support for complex data dependencies, validation conditions, and stream operations for handling noncontiguous formats.

Nail [7] was an early inspiration for Daedalus's design. Nail uses a single grammar to derive a parser, a generator and an internal representation for the parsed objects. It has support for dependent fields, manipulating the underlying parser stream, and can be extended by application specific code. Daedalus does not support generating serializers, but it extends the functionality of Nail, by adding features like a richer data description language, a type system, parameterized parsing rules, and support for working with bits.

Wuffs [33] is an imperative C-like programming language and libraries for parsing, serializing and deserializing binary formats. Its goals are industrial robustness, efficiency, and safety, addressing vulnerabilities such as buffer overflows, integer overflows and null pointer dereferences. Wuffs programs require explicit safety annotations and are transpiled into C to be easily integrated with C/C++ applications. In comparison, Daedalus attempts to provide a syntax and semantics in which format definitions are closer to parameterized grammars than low-level programs.

Parsley[52] is a Data Format Definition Language that extends Parsing Expression Grammars (PEG) to handle complex formats like PDF. Similar to Daedalus, it supports data dependencies and is able to generate memory-safe parsers. The parsing algorithm underlying Parsley is a formally verified packrat parser [9]. Parsley's parsing algorithm also computes a *proof-of-parse* certificate, which appears to be a unique feature in the field of parsing tools. Unlike Parsley, Daedalus supports

both ordered and non-ordered choice, along with explicit stream manipulation, and has the ability to compile grammars to executable code, for integration with an application.

13 CONCLUSIONS

Parsing is a classic problem in computer science. Nearly every undergraduate learns the fundamental theory and practice of parser construction. When developing Daedalus, we have sometimes encountered surprise that we are working on a long-'solved' problem. Yet, as of 2024, most parsers in deployment are constructed using insecure technologies and ad hoc methods. The result can be seen in CVE databases throughout the world, and in the growing security industry focusing on exploiting and/or mitigating parsing-related issues.

It is past time to build parser languages that are powerful enough to handle the complexities of real-world binary formats, performant enough to be deployed in practice, and secure enough to eliminate many parser vulnerabilities by default. Daedalus is our attempt to make progress on this problem, and this paper collects evidence that Daedalus can be used to solve parsing issues in a range of realistic binary-parsing scenarios.

There remains a great deal of interesting work to be done in format definition languages. One intriguing topic is the relationship between binary parsing and *schema conformance*. In PDF, we use Daedalus to define both the construction of COS objects from bits, and the schema constraining COS object values. In languages such as XML and JSON, these notions are separate, but in PDF they are (slightly) intertwined. At present, Daedalus forces us to intermingle concerns. But perhaps a better solution would allow us to define a pipeline of parsers, starting with binaries, building objects, applying schema rules, then maybe constructing higher-level semantic values. We have early ideas on what such a language would be like, but we encourage others to look at this problem.

Another intriguing topic is *parser correctness*[4]. In a sense, Daedalus is a specification language in itself, and we have designed it to be amenable to expert audit. But in practice, formats are defined by (1) human-readable prose specifications; (2) a set of existing implementations; and (3) a extant document corpus, which for PDF comprises billions of documents. For any non-trivial format, all of these will be inconsistent with each other and with basic security principles! To address this, we have built Daedalus as one component in a larger parser-correctness toolchain, with a complementary set of tools for *document understanding*, based on statistical inference and program analysis. We hope to submit a paper on this larger toolchain soon.

Finally, we are fascinated by *format analysis*. In a classic parser generator, the format is specified in a grammar which is highly amenable to analysis. In contrast, the type of ad hoc parsers that are prevalent in reality are just semi-structured programs in commodity languages such as C++, and are therefore very difficult to analyze. With Daedalus, we have a language that is both useful in these more complex environments and that has a clean syntax and semantics. This creates the opportunity to analyze Daedalus format descriptions much more easily than would be possible for previous parsers. Above, we have already discussed two types of analysis: our tool Talos which constructs conformant inputs for a Daedalus parser, and a detector for so-called *polyglot* files. Daedalus is open source, and we hope that others will use it as a platform for data format analysis.

14 DATA-AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in zenodo[39] and our repository [40].

[4]We *do not* mean the problem of correctly compiling Daedalus to a target language such as C++. This is interesting as an instance of compiler assurance / verification, but we think it is mostly orthogonal to parsing as a problem domain.

ACKNOWLEDGMENTS

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0076.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Pub. Co, Reading, Mass.
[2] Alfred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., USA.
[3] Ange Albertini, Thai P. Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. 2020. How to Abuse and Fix Authenticated Encryption Without Key Commitment. In *IACR Cryptology ePrint Archive*. https://api.semanticscholar.org/CorpusID:227231944
[4] Prashant Anantharaman. 2022. *Protecting Systems From Exploits Using Language-Theoretic Security*. Ph. D. Dissertation. Dartmouth College, USA. https://digitalcommons.dartmouth.edu/dissertations/80
[5] Andy Gill and Simon Marlow. Accessed: March, 2024. Happy. https://www.haskell.org/happy/
[6] ASN.1 Accessed: March 2024. ITU reference material for ASN.1. http://www.itu.int/en/ITU-T/asn1/.
[7] Julian Bangert and Nickolai Zeldovich. 2014. Nail: a practical tool for parsing and generating data formats. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 615–628.
[8] Karen Barrett-Wilt. 2021. The trials and tribulations of academic publishing – and Fuzz Testing. https://www.cs.wisc.edu/2021/01/14/the-trials-and-tribulations-of-academic-publishing-and-fuzz-testing/ Publication Title: UW. Madison Department of Computer Sciences.
[9] Clement Blaudeau and Natarajan Shankar. 2020. A verified packrat parser interpreter for parsing expression grammars. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 3–17. https://doi.org/10.1145/3372885.3373836
[10] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 111–122.
[11] Bryan O'Sullivan. Accessed: March, 2024. attoparsec: Fast combinator parsing for bytestrings and text. https://hackage.haskell.org/package/attoparsec
[12] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249
[13] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 3–11.
[14] Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on information theory* 2, 3 (1956), 113–124.
[15] Chris Dornan and Simon Marlow. Accessed: March, 2024. Alex User Guide. https://www.haskell.org/alex/
[16] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266
[17] Common Crawl Accessed: March 2024. Common Crawl. https://commoncrawl.org
[18] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18 (01 2008), 1–13.
[19] CVE-2022-27337 2022. CVE-2022-27337. https://nvd.nist.gov/vuln/detail/CVE-2022-27337
[20] CVE-2022-38784 2022. CVE-2022-38784. https://nvd.nist.gov/vuln/detail/CVE-2022-38784
[21] Daan Leijen and Erik Meijer. 2001. Parsec: direct style monadic parser combinators for the real world. https://api.semanticscholar.org/CorpusID:14373505
[22] David Peter. Accessed: March 2024. hyperfine: a command-line benchmarking tool. https://github.com/sharkdp/hyperfine
[23] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.* 3, ICFP (2019), 82:1–82:29. https://doi.org/10.1145/3341686
[24] DFDL Accessed: March 2024. DFDL (Open Grid Forum). https://www.ogf.org/ogf/doku.php/standards/dfdl/dfdl

[25] Iavor Sotirov Diatchki. 2007. *High-Level Abstractions for Low-Level Programming.* Ph. D. Dissertation. OGI School of Science & Engineering at Oregon Health & Science University.
[26] Olivier Dubuisson and Philippe Fouquart. 2001. *ASN.1: Communication Between Heterogeneous Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[27] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. 2004. Boltzmann Samplers for the Random Generation of Combinatorial Structures. *Combinatorics, Probability and Computing* 13, 4-5 (2004), 577–625. https://doi.org/10.1017/S0963548304006315
[28] Romain Edelmann, Jad Hamza, and Viktor Kunčak. 2020. Zippy LL(1) Parsing with Derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1036–1051. https://doi.org/10.1145/3385412.3385992
[29] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi
[30] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *SIGPLAN Not.* 39, 1 (jan 2004), 111–122. https://doi.org/10.1145/982962.964011
[31] Godmar Back. 2002. DataScript—A Specification and Scripting Language for Binary Data. In *Proceedings of the ACM Conference on Generative Programming and Component Engineering Proceedings (GPCE 2002), published as LNCS 2487*. Association for Computing Machinery, Pittsburgh, PA, USA, 66–77.
[32] Google. Accessed: March 2024. Protocol Buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf
[33] Google. Accessed: March 2024. Wuffs. Wrangling Untrusted File Formats Safely. https://github.com/google/wuffs
[34] Graham Hutton and Erik Meijer. 1996. *Monadic Parser Combinators.* Technical Report. University of Nottingham.
[35] Hammer 2019. Hammer: Parser Combinators for Binary Formats, in C. Yes, in C. What? Don't Look at Me like That. UpstandingHackers.
[36] Mike Hodge. 2020. Exploiting PHP Phar Deserialization Vulnerabilities: Part 1. *Keysight* (June 2020). https://www.keysight.com/blogs/tech/nwvs/2020/07/23/exploiting-php-phar-deserialization-vulnerabilities-part-1
[37] John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass.
[38] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. 2005. High-level Views on Low-level Representations. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. Tallinn, Estonia, 168–179.
[39] Iavor S. Diatchki, Mike Dodds, Harrison Goldstein, Bill Harris, David A. Holland, Benoit Razet, Cole Schlesinger, and Simon Winwood. Accessed: May 2024. Daedalus PLDI Artifact. https://doi.org/10.5281/zenodo.10966813
[40] Iavor S. Diatchki, Mike Dodds, Harrison Goldstein, Bill Harris, David A. Holland, Benoit Razet, Cole Schlesinger, and Simon Winwood. Accessed: May 2024. Daedalus Repository. https://github.com/GaloisInc/daedalus
[41] International Telecommunication Union. 1994. *Abstract Syntax Notation One (ASN.1): Specification of base notation.* Standard. ITU. https://handle.itu.int/11.1002/1000/14468
[42] ISO/TC 171/SC2. 2020. *ISO 32000-2:2020 (PDF 2.0).* Standard. International Organization for Standardization. https://www.iso.org/standard/75839.html
[43] Kaitai Accessed: March 2024. Kaitai Struct: declarative binary format parsing language. http://https://kaitai.io/
[44] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. 2010. The Next 700 Data Description Languages. *Journal of the ACM* 57, 2 (February 2010).
[45] Ravie Lakshmanan. 2023. Cybercriminals Using Polyglot Files in Malware Distribution to Fly Under the Radar. *The Hacker News* (January 2023). https://thehackernews.com/2023/01/cybercriminals-using-polyglot-files-in.html
[46] John Larmouth. 2000. *ASN.1 Complete.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[47] Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged parser combinators for efficient data processing. *ACM SIGPLAN Notices* 49 (2014), 637 – 653. https://api.semanticscholar.org/CorpusID:2341223
[48] Daniel Marjamäki. 2007. CppCheck: Static Analysis of C/C++ code. https://github.com/danmar/cppcheck
[49] Mark P. Jones. 1992. A Theory of Qualified Types. In *ESOP '92: European Symposium on Programming*. Rennes, France.
[50] Robert E McGrath. 2011. *Data Format Description Language: Lessons Learned, Concepts and Experience.* Technical Report. National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. https://core.ac.uk/display/4835576?utm_source=pdf
[51] Microsoft. Accessed: March 2024. Language Server Protocol. https://microsoft.github.io/language-server-protocol/
[52] Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean W. Smith. 2020. Research Report: The Parsley Data Format Definition Language. *2020 IEEE Security and Privacy Workshops (SPW)* (2020), 300–307. https://api.semanticscholar.org/CorpusID:219184197

[53] Nada Amin and Tiark Rompf. 2017. LMS-Verify: abstraction without regret for verified systems programming. *SIGPLAN Not.* 52, 1 (jan 2017), 859–873.
[54] nom Accessed: March 2024. nom, eating data byte by byte. https://github.com/rust-bakery/nom
[55] PicoHTTPParser Accessed: March 2024. PicoHTTPParser. https://github.com/h2o/picohttpparser
[56] Poppler Accessed: August 2023. Poppler Releases. https://poppler.freedesktop.org/releases.html
[57] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1465–1482. https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud
[58] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375.
[59] Sebastian Ullrich and Leonardo de Moura. 2021. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore). Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages.
[60] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook white paper* 5, 8 (2007), 127.
[61] Stephen C. Johnson. 1975. *Yacc: Yet Another Compiler-Compiler.* Technical Report. AT&T Bell Laboratories, Murray Hill, New Jersey 07974. PS1:15–1 – PS1:15–32 pages.
[62] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. 2022. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 31–45. https://doi.org/10.1145/3519939.3523708
[63] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.
[64] Tony Hannan. Accessed: March 2024. Haskell bson library. https://hackage.haskell.org/package/bson
[65] Mark Tullsen, William Harris, and Peter Wyatt. 2022. Strengthening Weak Links in the PDF Trust Chain. *LangSec Workshop* (2022). https://github.com/gangtan/LangSec-papers-and-slides/raw/main/langsec22/papers/Tullsen_LangSec22.pdf
[66] United States Department of Defense. 1993. *National Imagery Transmission Format (Version 2.0).* Standard. US DoD. https://earth-info.nga.mil/php/download.php?file=cib-2500a
[67] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (July 1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248 Conference Name: IEEE Transactions on Software Engineering.
[68] David Wheeler. 2001. Flawfinder. https://security.web.cern.ch/recommendations/en/codetools/flawfinder.shtml
[69] XpdfReader Accessed: March 2024. XpdfReader. https://www.xpdfreader.com
[70] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary F. Fernández, and Artem Gleyzer. 2007. PADS/ML: A Functional Data Description Language. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. Association for Computing Machinery, Nice, France, 77–83.