Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust

Geoffoy Couprie
Software Security & Architecture Consultant
Nantes, France
contact@geoffroycouprie.com

***Abstract*— The recently created language Rust has been presented as a safer way to write low level code, even able to replace C. Is it able to produce safe and efficient parsers? We show that Rust's features, like slicing, allow for powerful memory management, and that its type safety helps in writing correct parsers. We then study briefly how it can make streaming parsers, and how to provide better usability in a parsing library.**

I. INTRODUCTION

The facilities offered by the Rust language for safe memory manipulation[1] and low level code[2] make it well suited for parsers. As part of development at VideoLAN, we studied file and network parsers, the most dangerous part of the VLC media player application: almost all the reported vulnerabilities are related to demuxers, the network, video and audio format parsers[3]. This is due in part to the use of handwritten parsers in C and C++, languages that do not make it easy to write memory safe code[4]. This is also due to the complexity of supporting audio and video file formats: they are all very different, have unspecified bugs, undefined parts in their specs, and common (but unspecified) practices that must be followed. Those constraints make the parsers hard to follow, and their modifications bug prone.

While there is no project in the shorter or longer term to integrate Rust code directly in VLC media player, we decided to investigate its use in replacing some of the underlying C libraries used by VLC. It should be possible to make a Rust library with the same API as a C library, although we are not aware of any attempt at that project right now.

Parser combinators seemed a good way to implement safer parsers, as they allow for fast experimentation and reconfiguration.

The memory ownership system in Rust allows the developer to reference an immutable subset of an array everywhere in the code, without modifying or copying that array, so we set out to test a zero copy (as much as possible) parser in that language. Another interesting approach to test with parser combinators is to stream the parsing. Streaming parsers can have two modes. Push streaming parses data as soon as it comes from the source, is stateless and mainly driven by the source (useful for proxies and stoe.Pull streaming is driven by the data consumer, and has specific logic for state machine handling and seeking.

Since one of the goals was to write file and network parsers, we choose to work at the byte level, as much as possible, but we will see that Rust generics allow work on almost any data type.

II. RUST

Rust is a language designed for memory safety and concurrency, developed by Graydon Hoare since 2006 and sponsored by Mozilla since 2009. It is now at 1.0.0-alpha state (as of January 2015) and will reach the final 1.0 in the following months. It is the language of choice for Servo, Mozilla's browser research project, and it has been announced[5] that some components of Firefox will soon be written in Rust.

Developing in this language has been a challenge in the recent years, since a lot of features were added and removed, following research and tests. A large part of the language has been simplified and abstracted in libraries. Still, their approach of compiler development - do not fix bugs, fix bugclasses - has produced a solid platform to write robust code.

A. Error management

As one of its basic principles, Rust does not allow null pointers. It instead encourages the developer to employ other errors abstractions than integers, through enumeration types like *Option* or *Result*. They offer a semantic differentiation between values and errors, allow method chaining on values and error propagation. Contrary to most languages, where error condition checks are optional and made explicit by writing them, in Rust, errors are checked by default, and ignoring the error, by unwrapping the *Option* or *Result*, is made explicit.

A facility for exceptions exists, but is frowned upon for common error management, and left for undefined cases in code abstracted by threads. This approach joins the Erlang one, where letting code crash is preferable to handling every exceptional case. This keeps a manageable software base through the use of small, easy to restart processes.

B. Ownership and memory management

Memory management is handled at compile time, by deciding who owns a particular memory zone, and when it can be safely removed[6]. The compiler will refuse to build code if memory is used unsafely, or if it cannot decide which thread or code block owns a variable. It will insert allocation and deallocation code right where it is needed, and can use

DOI 10.1109/SPW.2015.31

Listing 1. Slice implementation

```
pub struct Slice<T> {
    pub data: *const T,
    pub len: usize,
}
```

any allocator the developer wishes to provide. In the past, a garbage collection system was included in the language, but improvements to the ownership system made it easier to rely only on the compiler, so the garbage collector was extracted in a library[7]. Most of the time, the Rust compiler is able to handle anything a garbage collector can do.

Data is defined as immutable by default, making it a lot easier for the compiler to reason about code, and for the developer to find the parts of the code where it can be mutated. Here again, we can compare common programming languages like C where mutability is the default, and immutability is made explicit by *const* tags, while in Rust, mutability is made explicit by *mut* tags. This gives assurance to the compiler and the developer than functions will not modify their input.

Moreover, immutable data can be safely transferred between threads, since it is guaranteed to have no concurrent modifications, so synchronization primitives like mutexes can be removed. In other cases, with mutable data, the ownership inference system will decide that no concurrent read and write will happen on a memory zone. When we still want concurrent modifications, some types provide mutexes and other primitives.

In some cases, memory has to be handled by hand, like when interfacing with specific C code, or in data structures like a doubly linked list, the language provides an *unsafe* tag to wrap that code. This creates a contract between the developer and the compiler: the compiler assumes that whatever is done inside the unsafe code part has been written carefully, and on that assumption, verifies the rest of the system.

By isolating the parts of the code that can mutate data or manipulate data unsafely, we can build complex code relying on the compiler's verification.

C. Zero copy support

A large part of array or list manipulations is done through slices, subsets of the data structure used. It is naturally usable with byte arrays and bit vectors, making it suitable for low level manipulations. The underlying implementation of slices for buffers is as displayed in the listing 1.

It contains a pointer to an array of the generic type *T*, and the length of the slice. With the compiler's work on ownership inference, we can pass around subsets of a buffer everywhere without copying the data, and without worrying that the original buffer would be deallocated. This property is a fundamental feature of nom.

Listing 2. Parser result enumeration

```
pub enum Needed {
  Unknown,
  Size(u32)
}
pub enum IResult<I,O> {
  Done(I,O),
  Error(Err),
  Incomplete(Needed)
}
```

III. PARSER COMBINATORS

Parser combinators are recursive descent parsers relying on small, reusable parsing components called recognizers, and combining them for more complex grammars. They are often implemented in functional languages, since the easy use of higher order functions and monadic constructions helps in constructing them, although they are not restricted to the functional languages[9]. They are well suited to implement context-free grammars, although with less capacity at error recovery than common parsers.

By writing small, reusable recognizers, we can make them easier to test for various inputs:

- what happens if the byte is outside of the ASCII range?
- what happens if the input array is empty?
- what happens if the input array is larger than requested?

Recognizers usually take an input array, and return either an error, or a tuple containing the recognized part, and the rest of the input. This tuple is often a cause of a lot of data copying between parsers.

The combinators take the output of the recognizers and decide what should be done, through simple constructions, like choosing the first correct result, chaining correct results, repeating the parser and assembling results in a vector.

The recognizers and combinators can be made completely generic, and work on various types of data, thus allowing designs where data is transformed as soon as possible to the desired output format.

Since they are also reusable, we can build partial parsers to handle just the part of the data we need, and separate it cleanly in the state machine.

Finally, the way recognizers are used tends to produce a code architecture matching closely the grammar, making it easy to review and modify.

IV. NOM, A PARSER COMBINATORS LIBRARY WRITTEN IN RUST

A. Implementation

Creating a parser combinator library is mostly straightforward. It needs a sum type holding the different results of a parser, that wen can define with generic input and output types, as seen in the Listing 2.

The *Incomplete(Needed)* field allows a parser to indicate, if it was not fed enough data, either that it needs a specific

Listing 3. Alphabetic character list parser

```
pub fn alpha(input:&[u8])
  -> IResult<&[u8], &[u8]> {

  for idx in 0..input.len() {
    if !is_alphabetic(input[idx]) {
      return Done(&input[idx..],
        &input[0..idx]
      )
    }
  }
  Done(b"", input)
}
```

Listing 4. Slice implementation

```
pub fn length_value(input:&[u8])
  -> IResult<&[u8], &[u8]> {
  let input_len = input.len();
  if input_len == 0 {
    return IResult::Error(0)
  }

  let len = input[0] as usize;
  if input_len - 1 >= len {
    return IResult::Done(
      &input[len+1..],
      &input[1..len+1]
    )
  } else {
    return IResult::Incomplete(
      Needed::Size(1+len as u32)
    )
  }
}
```

Listing 5. FlatMap implementation

```
pub trait FlatMap<I:?Sized,O:?Sized,
  N:?Sized> {
  fn flat_map<F:Fn(O) -> IResult<O,N>>
    (&self, f: F) -> IResult<I,N>;
}

impl<'a,R,S,T> FlatMapper<&'a[S], T>
  for IResult<R,&'a [S]> {
  fn flat_map<F:Fn(&'a[S])
    -> IResult<&'a[S],T>>
    (&self, f: F) -> IResult<&'a[S],T> {
    match self {
      &Error(ref e) => Error(*e),
      &Incomplete(Needed::Unknown) => {
        Incomplete(Needed::Unknown)
      },
      &Incomplete(Needed::Size(ref i)) => {
        Incomplete(Needed::Size(*i))
      },
      &Done(ref i, ref o) => match f(*o) {
        Error(ref e) => Error(*e),
        Incomplete(Needed::Unknown) =>  {
          Incomplete(Needed::Unknown)
        },
        Incomplete(Needed::Size(ref i2)) => {
          Incomplete(Needed::Size(*i2))
        },
        Done(_, o2) => Done(*i, o2)
      }
    }
  }
}
```

amount of data, or that it needs more data but does not know how much.

We can then define a parser as a function taking an input, and giving a *IResult*. For a simple function matching alphabetic characters, we do not need to return any errors: it will just return an empty output (cf listing 3).

Some basic parsers with error return can be defined, such as the length-value pattern, frequent in network formats (cf listing 4).

Note that data is never copied. By using the slicing API for vectors, we can get a sized reference to a subset of the vector, and use it throughout the code.

1) Combining parsers: After defining those functions, we need ways to assemble them. The first function we need for this is *flat_map*. Such a *flat_map* function allows us to apply a parser on the result of another one, without rewrapping it in the *IResult* type (cf listing 5).

We define it as a function implemented for a *IResult<I,O>*, taking as argument another function working on the *O* type and returning another result. Because of the way the type system is made, we have to define it for a lot of *<O,N>* type combinaisons.

The tokens preceded by apostrophes are indications about data lifetime for the compiler. Basically, all data tagged with the lifetime *'a* will stay allocated at the same time (or more if allowed).

In this case, we have to define two lifetimes, one for the output of the *IResult*, and one for the remaining input. Note that the flat_map function returns, at least in the case of a type *T* that does not require a lifetime (on the contrary of an array slice), a result that only depends on the 'a lifetime. This is why the compiler allows data to be transmitted from the beginning of the parsing chain to the end without copying it: the compiler knows that the data at the end of the parsing chain comes from the same buffer and will not deallocate it until it is not needed anymore.

We also define a *map* operation on *IResult*, to work directly with usual functions that return *Option* or *Result* types. Those methods are essential, since we can reuse data transformation functions already provided by Rust's libraries (cf listings 7

Listing 6. FlatMap usage

```
Done((),b"abcd").flat_map(|data| {
  println!("data: {:?}", data);
  Done(data,())
});
```

Listing 7. FlatMapOpt implementation

```
pub trait FlatMapOpt<I,O,N> {
  fn map_opt<F: Fn(O) -> Option<N>>
    (&self, f: F) -> IResult<I,N>;
  fn map_res<P,F: Fn(O) -> Result<N,P>>
    (&self, f: F) -> IResult<I,N>;
}

impl<'a,'b,R,S,T>
  FlatMapOpt<&'b[R],&'a[S],T>
  for IResult<&'b [R],&'a [S]> {
  fn map_opt<F:Fn(&'a[S]) -> Option<T>>
    (&self, f: F) -> IResult<&'b [R],T>
    match self {
      &Error(ref e) => Error(*e),
      &Done(ref i, ref o) => match f(*o)
        Some(output) => Done(*i, output),
        None         => Error(0)
      }
    }
  }

  //fn map_res<U, F: Fn(&'a[S])
    -> Result<T,U>>
  (&self, f: F) -> IResult<&'b [R],T> ...
}
```

and 8).

2) Facilities for parser definitions: One of the issues with Rust is that the developer has to be really precise about the types used, and there is very limited type inference. The code often ends up littered with type definitions, moreover with highly generic code like this library. Fortunately, the language offers a macro system with limited support for the language construction: it can recognize arguments such as types, identifiers, expressions and a few others. This permits the definition of generic parser combinators.

The code in listing 9 defines a macro for optional usage of a parser. It takes a name for the parser, an input type, an output type, and another parser to apply. That way, we can easily define an optional parser from another one, defined

Listing 8. FlatMapOpt usage

```
Done((),b"abcd").map_res(|data| {
  str::from_utf8(data)
});
```

Listing 9. Optional macro implementation

```
macro_rules! opt(
  ($name:ident<$i:ty,$o:ty> $f:ident)
  => (
    fn $name(input:$i)
      -> IResult<$i, Option<$o>> {
      match $f(input) {
        IResult::Done(i,o) => {
          IResult::Done(i, Some(o))
        },
        _                  => {
          IResult::Done(input, None)
        }
      }
    }
  )
);
```

Listing 10. tag and optional macros usage

```
tag!(txt_parser b"abcd");
opt!(opt_txt_parser<&[u8], &[u8]> txt_parser);
 {
assert_eq!( Done((),b"abcdef")
  .flat_map(opt_txt_parser),
  Done(b"ef", Some(b"abcd")) );
assert_eq!( Done((), b"bcdefgh")
  .flat_map(o), Done(b"bcdefg", None) );
```

with a macro itself (cf listing 10).

Most of the useful combinations, like the *chain* or *alt* parsers are currently defined as macros themselves. The macro usage allows flexibility in the syntax and helps in representing interesting patterns, like filling a structure with the chain macro (cf listing 11).

B. Streaming parsers

There are two sorts of streaming parsers. The **push parsers** will parse data as soon as it arrives. Their goal is to be fast, not smart. In nom, they were implemented through the *Producer* trait, managing files or in-memory buffers (cf listing 12).

The other type of streaming parser is more interesting: a **pull parser** takes decisions based on the data it has seen, can decide wether it needs more input, and can have support for seeking. This is will more closely represent a protocol's state machine, or an application looking for the data it needs through a file. It was implemented in nom with the *Consumer* trait (cf listing 13).

A streaming parser, to be efficient, must be able to work on incomplete data, and restart from there once new data is available. The difficulty is then in two ways: recognizing the difference between a parsing error and incomplete data, and encoding it in the state machine.

Listing 11. chain macro usage

```
// matches a "key = value" in a INI file
// and returns it as a tuple of strings
chain!(key_value  <&[u8],(&str,&str)>,
    key: parameter_parser ~
         space?           ~
         equal            ~
         space?           ~
    val: value_parser     ~
         space?           ~
         comment_body?    ~
         line_ending?     ,

    ||{(key, val)}
);
```

Listing 12. Producer trait

```
pub enum ProducerState<O> {
  Eof(O),
  Continue,
  Data(O),
  ProducerError(Err),
}


pub trait Producer {
  fn produce(&mut self)
    -> ProducerState<&[u8]>;
  fn seek(&mut self,
    position:SeekFrom)
    -> Option<u64>;
}
```

This is the part where the zero copy claim gets complicated. If new input must be added to the current buffer it requires a reallocation, which will slow the parsing. And parsing again the whole buffer will waste a lot of time.

Implementing efficient streaming parsing requires interventions in different parts of the code. In the parsers, but also at the two ends of the parsing pipeline, the producers (managing files or in-memory buffers) and the consumers (handling the results and the state machine).

1) IResult::Incomplete: While it is a special case that most parsers will not handle, returning *Incomplete* in some cases allows the parser to ask for more input, and have that message pass through all the combinators to the producer or consumer handling the data.

Multiple implementations were tested. The first idea was to return a closure in *Incomplete*, like **Attoparsec**'s partials, but it proved too difficult for the ownership system to decide what to do with the data captured, and failed compilation.

The second idea was to have two fields in *IResult*: *IncompleteUnknown* and *Incomplete(size)*, to represent the need for more data, with or without a specific quantity in mind. This had two issues: four fields to match in combinators made the compilation too long, as it is heavily based on macros, and this made too much boilerplate to write for parser and consumer implementors.

Listing 13. Consumer trait

```
pub enum ConsumerState {
  Await(
    usize,     // consumed
    usize      // needed buffer size
  ),
  Seek(
    usize,     // consumed
    SeekFrom, // new position
    usize      // needed buffer size
  ),
  Incomplete,
  ConsumerDone,
  ConsumerError(Err)
}

pub trait Consumer {
  fn consume(&mut self, input: &[u8])
    -> ConsumerState;
  fn end(&mut self);

  fn run(&mut self,
    producer: &mut Producer);
}
```

The chosen solution is the one described in the *IResult* definition in listing 2. *Incomplete* contains a sum type that can be a size, or *Unknown*. This makes it easy to compile in a reasonable time, parser developers can ignore the value with pattern matching on *Incomplete(_)*, and developers with more specific needs can obtain the value.

While the closure solution would have avoided some redundant parsing of the input data, we will see that it is not necessary in practice.

2) Seeking producer: While the producer does not need to have much logic to manage data, making it able to signal that more data is available can be useful to the consumer. Making them seekable when possible, backward and forward for files and memory buffers, forward for network packets, allows an interesting speed improvement and a reduction of memory consumption. We do not need to load the whole buffer, since we can just jump to the part we want. Another idea to improve producer performance is to make their chunk size reconfigurable: after header parsing, the state machine will often know how much more data it needs.

3) Partial parsing in the consumer: A consumer is written by implementing the *Consumer* trait and its methods *consume* and *end*. Then calling the *run* method with a producer will start the parsing pipeline. The *run* method handles data aggregation from the producer and passes it to the *consume* method which will only care about parsing

and state modification. It is easily represented as a switch between states, and applying the corresponding parsers, as seen in that example consumer implementation (cf listing 14).

There are two things to note here. Since *run* handles the ”plumbing”, it will not load more data than needed, and will command the producer according to the returned *ConsumerState*. The consumer only needs to indicate how much was consumed and how much it needs, or where it can be in the stream, and *run* will handle everything to avoid unnecessary data copying. The other important point is that we are not limited to parser states indicated in the specification of a protocol or format. By making multiple intermediary state, we can avoid parsing large parts of the data multiple times if the input buffer is not large enough: the parser will only return *Incomplete* on the most recent buffer slice, and we keep the previously parsed data. Finally, while switch-based state machines are easy to implement, it is known that they make it easy to bypass parser states[10].

C. User experience

One of the goals of that project is to make it easy to write complex parsers. This required work on multiple parts.

1) Parser combinators: Choosing parser combinators as a basis proved useful, since writing parsers incrementally is easier than trying to target the whole format at once. Most of the parsers in nom and in its examples are unit tested. This made experimentation a lot easier, since most of the code was never broken silently.

Property based testing could be investigated, to make those parser tests exhaustive.

2) Syntax: Earlier versions of the example parsers were presented regularly to developers, to get feedback on readability. The decisive factor for positive feedback was the introduction of complex macros like the *chain* one, to abstract all of the boilerplate.

The code still contains a lot of type definitions, which do not help readability but are required by Rust.

The Rust compiler has a plugin facility allowing AST manipulations. While those plugins are harder to write than macros, they are more flexible, and could be used to make more readable parsers.

3) State machine representation: Representing a state machine in a usable way is a complex problem. While state transitions are easy to encode (as a switch, a graph or a matrix), interacting with the state machine through an API and keeping the code easy to read is hard. A method of the API has to ask for a specific ”path” in the state machine, and gather the results in the terminal node (harder to do in a typesafe way).

The currently planned approach is first to make state machines and their transitions composable. This will allow a same parsing path to be reused between multiple origin and terminal nodes. Terminal nodes will hold a value of a specific type, which will guarantee type safety when writing code to interface state machines.

Listing 14. Consumer trait implementation

```
impl Consumer for TestConsumer {
  fn consume(&mut self, input: &[u8])
  -> ConsumerState {
    match self.state {
      State::Beginning => {
        match om_parser(input) {
          Error(a)       => {
            ConsumerState::ConsumerError(a)
          },
          Incomplete(_) =>{
            ConsumerState::Await(0, 2)
          },
          Done(_,_)      => {
            self.state = State::Middle;
            ConsumerState::Await(2, 3)
          }
        }
      },
      State::Middle    => {
        match nomnom_parser(input) {
          Error(a)          => {
            self.state = State::End;
            ConsumerState::Await(0, 7)
          },
          Incomplete(_)     => {
            ConsumerState::Await(0, 3)
          },
          Done(i,noms_vec) => {
            self.counter = self.counter +
              noms_vec.len();
            ConsumerState::Await(input.len() -
            i.len(), 3)
          }
        }
      },
      State::End        => {
        match end_parser(input) {
          Error(a)       => {
          ConsumerState::ConsumerError(a)
          },
          Incomplete(_) => {
            ConsumerState::Await(0, 7)
          },
          Done(_,_)      => {
            self.state = State::Done;
            ConsumerState::ConsumerDone
          }
        }
      }
    }
  }

  fn end(&mut self) {
    println!("counted {} noms",
    self.counter);
  }
}
```

The last step is to write the glue that will represent the states, their transitions through parsers, and the type that can be held by an ending state. Making an API available consists in describing a named method which will set an origin state, and wait for data of a specific type at a specific terminal state. The origin state should be directly reachable from the previous terminal state, and an arrival at a different terminal state will be treated as an error. Of course, we make no guarantee about the function terminating.

This state machine representation will most probably be implemented as another macro system.

4) Example parsers: Three example parsers have been provided with the code, one of them for the INI file format, another one for the MP4 fie format.

While the INI format was useful to test different parser combinations, the MP4 parser was the most interesting to test the producers and consumers, since it had to work efficiently on files from a few kilobytes to multiple gigabytes.

V. Performance

Some limited benchmarks were done to compare nom's performances to more well known solutions, **hammer** and **Attoparsec**. As always with benchmarks, they have to be considered with skepticism. I am not an expert in writing parsers with the other solutions and they probably could be improved. As an example, the C parser with hammer did not use any specific parser backend, another one may provide better performance.

On the other side, the measured parser in nom does not use any seeking or state machine optimization, only the most nave recursive parsing. This forces all the parsers to load all the data in memory before reading it, which make it easier to compare them. There is another version of the MP4 parser available, that can handle seeking, and parses large files easily.

This benchmark tests the ability to read two MP4 files, parse the filetype header and go through all the other boxes in the file, and measures the speed in *nanoseconds per iteration*. This test was realized on a late 2013 Macbook Pro (CPU quad core 2,3 GHz Intel Core i7). Each benchmark has been done with and without compiler optimizations. The code is available for reproduction[11].

TABLE I

Benchmark results without optimizations

	small.mp4 (375kB)	bigbuckbunny.mp4 (5.3 MB)
hammer	32807 ns/iter (+/- 91 ns)	28115 ns/iter (+/-82 ns)
attoparsec	1699 ns/iter (+/- 137 ns)	1601 ns/iter (+/- 105 ns)
nom	9619 ns/iter (+/- 1538 ns)	9083 ns/iter (+/- 2193 ns)

The hammer parser may not be completely optimized, and had some memory leaks that will be fixed it in a future release. It seems attoparsec does not get much faster after optimizations, even using llvm as the compilation backend, while nom gets a big performance improvement. The code has not been profiled, so we cannot verify that memory access patterns and slicing are the cause for the speedup. Still, we were able to measure that the nom parser process stays at 760 kB of memory while continuously parsing the two files, and the attoparsec parser stays at 7.5 MB of memory. For the nom parser, this is much less than the size of the files, since it only manipulates slices, and does not load the data unless it is needed.

TABLE II

Benchmark results with optimizations

	small.mp4 (375kB)	bigbuckbunny.mp4 (5.3 MB)
hammer	32424 ns/iter (+/- 47 ns)	26523 ns/iter (+/-77 ns)
attoparsec	1548 ns/iter (+/- 74 ns)	1476 ns/iter (+/- 69 ns)
nom	240 ns/iter (+/- 56 ns)	195 ns/iter (+/- 69 ns)

Of the three tested parsers, the easiest to write was the attoparsec one, then the nom one, then hammer.

VI. Conclusions

Rust has proven useful as a language to write parsers. Its powerful macro system allow complex code to be written naturally and safely. The zero-copy capability seems to make the parsers quite fast.

Future work on this project would consist mainly in writing more parsers, to tune its behaviour and fix inevitable bugs, integrating property based testing and fuzzing (random fuzzing, and state machine aware fuzzing), and implementing the state machine representation described precedently. Adding smarter parser backends like hammer would definitely be an interesting project, too.

References

[1] Nicolas Matsakis,, Guaranteeing Memory Safety in Rust, video: https://air.mozilla.org/guaranteeing-memory-safety-in-rust/ slides: http://fr.slideshare.net/nikomatsakis/guaranteeing-memory-safety-in-rust-39042975

[2] Unsafe and Low-Level Code, the Rust book, https://doc.rust-lang.org/book/unsafe.html

[3] https://www.videolan.org/security/

[4] Periklis Akritidis, Practical memory safety for C, UCAM-CL-TR-798

[5] Steve Klabnik, The Story of Rust, FOSDEM 2015

[6] Ownership, the Rust book, https://doc.rust-lang.org/book/ownership.html

[7] Patrick Walton, Removing Garbage Collection From the Rust Language, http://pcwalton.github.io/blog/2013/06/02/removing-garbage-collection-from-the-rust-language/

[8] Graham Hutton and Erik Meijer, Monadic Parser Combinators, 1996

[9] https://github.com/Geal/Pharsec

[10] SKIP-TLS attack, https://www.smacktls.com/#skip. Example patch for mono at https://github.com/mono/mono/commit/1509226c41d7

[11] https://github.com/Geal/nom_benchmarks. The last commit at the time of the benchmark was b3034a5c857b702ed1f2b8de89166937caeff192