Sequoia: A New OpenPGP Implementation in Rust

An Experience Report

Neal H. Walfield and Justus Winter

Rust Fest, Nov. 24, 2018

`https://sequoia-pgp.org/talks/2018-11-rust-fest`

Sequoia

- A new OpenPGP implementation in Rust
  - First commit: October 16, 2017
- Motivation
  - GnuPG is hard to modify
    - Code and API grew organically over 21 years
    - Lack of unit tests
    - Tight component coupling
  - Many developers unsatisfied with GnuPG's API
  - Rust is memory safe
  - GnuPG can't be used on iOS due to GPL

Who We Are

p≡p

- Neal, Justus, Kai
  - Former GnuPG developers (2–2.5 years at g10code)
  - At p≡p since Fall 2017
- Funding
  - p≡p (primary)
  - Wau Holland Stiftung (secondary)
  - (We're actively looking to diversify funding base!)

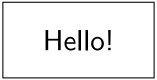
OpenPGP

- Encryption and Data Authentication & Integrity Standard
  - RFC 4880
- Not just for email...
  - Package Signing
  - Commit Signing
  - Document Signing (integrated in LibreOffice)
  - Backups, Archives
  - Encrypted Storage in the Cloud
  - Encrypted Sneaker Net
  - Password Manager
  - Remote Authentication (e.g., ssh agent)

Packet-Based Format

- An OpenPGP message is composed of packets:
    - Literal Data Packet
    - Signature + One Pass Signature Packet
    - Compression Container
    - Symmetrically Encrypted Data Packet (SEIP)
    - Public-Key Encrypted Session Key Packet (PKESK)
    - ...

An OpenPGP Message

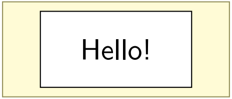


Hello!

- Some Data
- Encapsulate in an OpenPGP packet
- Sign it
- Encrypt it

- Looks like a pipe

An OpenPGP Message

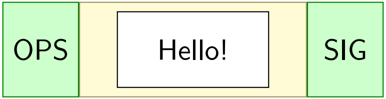


Literal Data

- Some Data
- Encapsulate in an OpenPGP packet
- Sign it
- Encrypt it

- Looks like a pipe

An OpenPGP Message

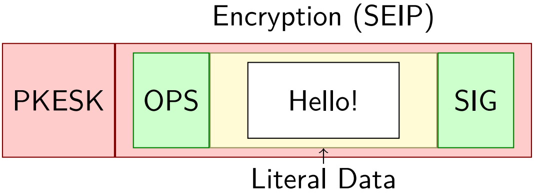


Literal Data

- Some Data
- Encapsulate in an OpenPGP packet
- Sign it
- Encrypt it

- Looks like a pipe

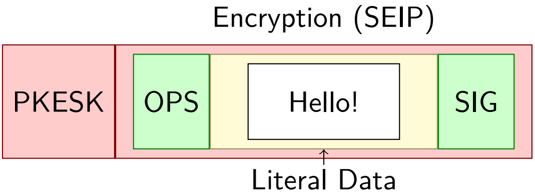
An OpenPGP Message



- Some Data
- Encapsulate in an OpenPGP packet
- Sign it
- Encrypt it

- Looks like a pipe

An OpenPGP Message



- Some Data
- Encapsulate in an OpenPGP packet
- Sign it
- Encrypt it

- Looks like a pipe

Aside: OpenPGP Messages Also Used for Key Exchange

- Public-Key Packet
- Public-Subkey Packet
- User ID Packet
- . . .

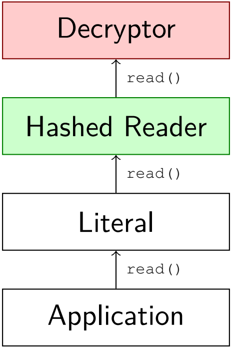
Introduction

- Sequoia
- OpenPGP

Challenges

- Processing a Pipeline
- A Generic Reader Stack
- A Better Parser Interface
- Callbacks that Save State
- Smuggling Failures in `io::Error`s

Challenge: Processing a Pipeline



- Create a stack of readers!
- Readers can also be used to deal with framing
    - Enforce packet boundaries
    - Chunked encoding

Depth-First Traversal of an OpenPGP Message

- Visitor Pattern
- Uses the stack

Idea

```
impl<R: Reader> CompressedData<R> {
    fn parse(mut reader: R) {
        if let Some(algo) = reader.read_u8() {
            let reader = match algo {
                1 => CompressedData { inner: reader },
                ...
            };
            parse(reader);
        }
    }
}

fn parse<R: Reader>(mut reader: R) {
    if let Some(tag) = reader.read_u8() {
        match tag {
            8 => CompressedData::parse(reader),
            ...
        }
    }
}
```

```
 error: reached the recursion limit while instantiating
↪`parse::<CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<CompressedData<⌋
↪CompressedData<CompressedData<GenericReader<⌋
↪std::fs::File> > > > > > > > > > > > > > > > > > > > > > > > > > > > > > > > > > > > >
```

Lessons

- Generics can result in a lot of invisible types
- There is no way to articulate a base case to the compiler
  - Can limit recursion at run-time
  - No way to express this to the compiler
- $\implies$ need dynamic dispatch

Challenge: A Generic Reader Stack

- Need an `into_inner()` for trait objects
- But trait objects are unsized!
- Can use a special version of `self`:

  ```
  pub trait BufferedReader {
      fn into_inner<'a>(self: Box<Self>)
          -> Option<Box<BufferedReader + 'a> >
          where Self: 'a;
  }
  ```

- Need an `Option` to handle the base case

`into_inner` and a Transparent Forwarder

- Working with boxed objects is ugly
- Often requires unnecessary boxing and unboxing
- Can use a transparent forwarder:

```
impl<'a> BufferedReader for Box<BufferedReader + 'a>
{
    fn buffer(&self) -> &[u8] {
        self.as_ref().buffer()
    }

    ...
}
```

- Can now pass a `Box<BufferedReader>` wherever a `BufferedReader` is needed

But, this creates a linked-list with `into_inner()`

```
impl<R: BufferedReader> BufferedReaderLimitor<R> {
    pub fn new(reader: R) -> Self {
        ...
    }
}
impl<R: BufferedReader> BufferedReader
    for BufferedReaderLimitor<R>
{
    fn into_inner<'a>(self: Box<Self>)
        -> Option<Box<BufferedReader + 'a>>
        where Self: 'a {
        Some(Box::new(self.reader))
    }
}
```

- `into_inner` boxes the inner reader
- The inner reader is of type `R`
- If R is a `Box<BufferedReader>`, we now have two boxes
- Each `new` / `into_inner` adds another box!
- The constructor needs to take a Box<BufferedReader> to avoid this

Challenge: A Better Parser Interface

- Using the visitor pattern requires callbacks
- Would prefer an iterator-like API
- OpenPGP messages can be huge
    - Requires streaming operations

An Iterator Interface

```
let pp = PacketParser::from_reader(r).unwrap();
for packet in pp.iter() {
    eprintln!("{:?}", packet);

    if let Packet::LiteralData(l) = packet {
        // This doesn't work!
        io::copy(&mut l, &mut io::stdout())
            .expect("Decryption failed");
    }
}
```

- Doesn't allow streaming
  - The returned item can't reference the original object
  - But the `reader` has to stay embedded in the `PacketParser` to get the next packet!
- Flattens tree structure

An Iterator-like Interface

```
let mut ppr = PacketParser::from_reader(r).unwrap();
while let PacketParserResult::Some(mut pp) = ppr {
    // Streaming operations...
    match pp.packet {
        Packet::Literal(_) =>
            io::copy(&mut pp, output)?,
        ...
    }

    let (packet, ppr_) = pp.recurse()?;
    ppr = ppr_;

    // We own the packet & can save it without copying.
    match packet {
        ...
    }
}
```

- Three phases
- Similar enough to Rust's Iterator API to be familiar
- If we don't want to recurse into a container, can use `next`

Challenge: Callbacks that Save State

- Often want to collect some state, but how to propagate it?
- In C:

```
struct callback_state {
  ...
}

void callback(void *cookie) {
  struct callback_state *state = cookie;
  ...
}

void g() {
  struct callback_state state;
  function(&state, callback);
}
```

- In Rust, don't use a cookie, use a trait!

Example

```
trait CallbackHelper {
    fn callback(&mut self);
}

fn function<CB: CallbackHelper>(cb: &mut CB) {
    cb.callback()
}

struct Callback { }

impl CallbackHelper for Callback {
    fn callback(&mut self) {
        println!("Hello, world!");
    }
}

fn main() {
    function(&mut Callback { });
}
```

Challenge: Smuggling Failures in `io::Errors`

- We use `failures`
- We also implement general purpose traits (e.g., `io::Read`)
- Failures can be returned via an `io::Error` using `failure::compat`!

Converting a Failure to an `io::Error`

```
match result {
    Ok(r) => Ok(r),
    Err(e) => match e.downcast::<io::Error>() {
        // An io::Error.  Pass as-is.
        Ok(e) => Err(e),
        // A failure.  Create a compat object & wrap it.
        Err(e) =>
            Err(io::Error::new(io::ErrorKind::Other,
                               e.compat())),
    },
}
```

Recovering the Failure

```
let result = io::copy(&mut verifier, output)
    .map_err(|e| if e.get_ref().is_some() {
        // Wrapped failure::Error.  Recover it.
        failure::Error::from_boxed_compat(
            e.into_inner().unwrap())
    } else {
        // Plain io::Error.
        e.into()
    })?;
```

- `verifier` is a custom reader
- Using this pattern, it is still able to use rich errors!

Interested in Sequoia?

- Sequoia's low-level API is 98% feature complete
  - ...including a C FFI
  - Port to p≡p Engine nearly complete
    - About 60% as much code as version using GPG
  - gpgv replacement
  - New keyserver implementation
  - Experiments porting other software
- First release was...
- Join us!
  - (We're looking to hire people to help with Android and iOS integration.)

Interested in Sequoia?

- Sequoia's low-level API is 98% feature complete
  - . . . including a C FFI
  - Port to p≡p Engine nearly complete
    - About 60% as much code as version using GPG
  - gpgv replacement
  - New keyserver implementation
  - Experiments porting other software
- First release was. . . right now!
- Join us!
  - (We're looking to hire people to help with Android and iOS integration.)

Andy Wright, CC BY 2.0

Summary

`https://sequoia-pgp.org`

- Sequoia is a new OpenPGP implementation
- User-centric development
- Strong focus on security
- Portable & highly integrated
- Low-level API is already usable
- Join us on…
  - irc: #sequoia on Freenode
  - mailing list: devel@sequoia-pgp.org
  - gitlab: `gitlab.com/sequoia-pgp/`

Sequoia by steve lyon, CC BY-SA 2.0