rustls

a modern tls stack in rust

approach

make something that is easy and safe for 90% of uses

    but perhaps might not ever cater to the remaining 10%

"if in doubt, leave it out"

don't require configuration for 90% of such uses

    sane, well-regarded defaults

pay attention to safety and ergonomics of API

"The most dangerous code in the world: validating SSL certificates in non-browser software" - Boneh et al, ACM CCS'12

"Our main conclusion is that *SSL certificate validation is completely broken in many critical software applications and libraries*" ...

"For the most part, the actual SSL libraries used in these programs are correct" ...

"The primary cause of these vulnerabilities is the developers' misunderstanding of the numerous options, parameters, and return values of SSL libraries" ...

"many SSL libraries are unsafe by default, requiring higher-level software to correctly set their options, provide hostname verification functions [...]"

"APIs should present high-level abstractions to developers, such as “confidential and authenticated tunnel,” as opposed to requiring them to explicitly deal with low-level details such as hostname verification"

https://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf

rust is a nice language for this work

has a memory- and concurrency-safe subset

- most SSL/TLS vulnerabilities are construction errors, often catastrophic in a memory-unsafe language
- concurrency-unsafety is also a major source of software defects [citation needed]

excellent bi-directional FFI support

no runtime overhead (no GC), so suitable for embedded usage too

typically as fast as C, sometimes faster

current TLS feature list

TLS1.2 and TLS1.3-draft-18 (+ drafts 19, 20)

Client and server end

AES-128-GCM, AES-256-GCM, chacha20poly1305 suites (9 in total)

Forward secrecy, always

Server authentication, always

Client authentication, optional

All kinds of resumption

current TLS non-feature list

SSL2, SSL3, TLS1.0, TLS1.1

Renegotiation (it's nasty, broken several times, killed in TLS1.3)

RC4, DES, Triple-DES, discrete log DH, EXPORT suites, …

Suites not providing forward secrecy

No support for TLS1.3 0RTT data as yet (scary! needs extreme care)

detailed TLS non-feature list

feature	why unsupported
RC4	biases ruin confidentiality
DES/3DES	block size too small
export suites	deliberately bad
CBC suites	IV chaining, mac-then-encrypt, ...
renegotiation	no binding between sessions; underspecified

feature	why unsupported
compression	unsafe with mix of secret and attacker-controller data
MD5/SHA1	not collision resistant
static RSA KX	no forward secrecy
discrete log DH	cannot be negotiated in TLS1.2, poor performance

about TLS1.0/1.1

no obviously safe ciphersuites

- closest has bad design flaws
- countermeasures possible, but ugly hacks
- RFC7366 encrypt-then-MAC could fix this, but in practice mis-designed

long tail of old/misconfigured servers

early stages of death? (PCI from June 2018, Apple ATS, Cloudflare 'Require Modern TLS')

internals

msgs/*.rs: things that can de/serialise all the TLS types into convenient rust structs

```
Message { typ: Handshake, version: TLSv1_2, payload: Handshake(HandshakeMessagePayload
{ typ: ServerHello, payload: ServerHello(ServerHelloPayload { server_version: TLSv1_2,
random: Random([87, ..., 186]), session_id: SessionID, cipher_suite:
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256, compression_method: Null, extensions:
[RenegotiationInfo(PayloadU8([])), ServerNameAcknowledgement,
SessionTicketAcknowledgement, ECPointFormats([Uncompressed])] }) }) }
```

internals

server_hs.rs/client_hs.rs: state machine for server/client

- complexity hotspot (~1300/~1500 lines each).
- ripe for refactoring: would like to express correct dataflows in type system
- consists of functions of form:

```
fn handle_server_hello(sess: &mut ClientSessionImpl, m: Message) -> Result<&'static State, TLSError>
{ … }

pub static EXPECT_SERVER_HELLO: State = State {
    expect: Expectation {
        content_types: &[ContentType::Handshake],
        handshake_types: &[HandshakeType::ServerHello],
    },
    handle: handle_server_hello,
};
```

testing

automated test suite made up of:

- integration tests against openssl and some public web servers
- api-level tests
- unit tests of library internals
- 'bogo' - the BoringSSL test suite, built from the golang TLS stack
    - hugely comprehensive and impressive
    - found some good bugs in rustls
- 'TryTLS' - python test suite concentrating on common implementation errors
- performance benchmarks

currently ~95% line coverage

performance

date	send speed (Gbps per core)	recv speed (Gbps per core)
2016-09-04	1.5	2.3
2016-09-20	2.0	6.5
2016-09-27	4.4	6.2
2017-01-26	15.9	15.0
2017-05-10	22.7	21.4

all measurements taken on the same i5-6500 at 3.2GHz
suite: `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`

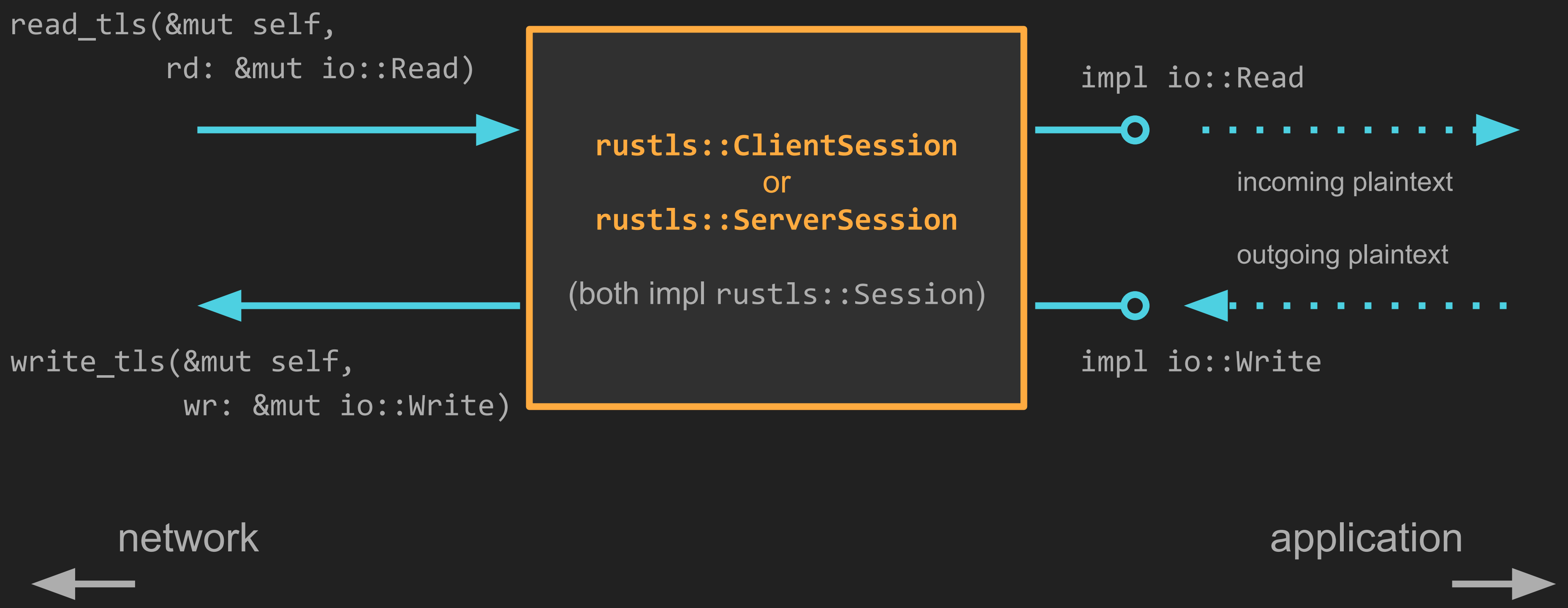
future work

- work out what to do with TLS1.3 0RTT data, if anything
- maybe support TLS1.1?
    - needs care, but not a great deal of work
- OCSP/SCT stapling support; in progress
- PSK support
- express TLS state machine in type system
- third party audit

thanks

Repo:	https://github.com/ctz/rustls
Test server:	https://rustls.jbp.io/
Twitter:	@jpixton
Mail:	jbp@jbp.io

api


read_tls(&mut self,
rd: &mut io::Read)
rustls::ClientSession
or
rustls::ServerSession
(both impl rustls::Session)
write_tls(&mut self,
wr: &mut io::Write)
impl io::Read
incoming plaintext
outgoing plaintext
impl io::Write
network
application

api

```
pub trait Session: Read + Write + Send {

   fn read_tls(&mut self, rd: &mut Read) -> Result<usize, io::Error>;


   fn write_tls(&mut self, wr: &mut Write) -> Result<usize, io::Error>;


   fn process_new_packets(&mut self) -> Result<(), TLSError>;

(...)

}
```

internals

- cipher.rs: TLS record layer encryption. a few implementations of these traits:

```
pub trait MessageDecrypter : Send + Sync {
    fn decrypt(&self, m: Message, seq: u64) -> Result<Message, TLSError>;
}

pub trait MessageEncrypter : Send + Sync {
    fn encrypt(&self, m: BorrowMessage, seq: u64) -> Result<Message, TLSError>;
}
```

api

to make a `rustls::ClientSession` you need a `rustls::ClientConfig` and the server's dns name

- one `ClientConfig` per process, typically
- contains root certificates
- plus a trait impl that persists data between sessions for resumption

similarly for `rustls::ServerSession` and `rustls::ServerConfig`

- `ServerConfig` contains a trait impl that chooses what certificate chain & private key to use for session

api usability

a `rustls::Session` buffers outgoing data if needed

- so you can send your HTTP request before the TLS handshake completes
- data only sent if handshake and authentication successful
- data sent in same flight as last message in handshake

the 'bring your own IO' interface has moderate usability cost

- allows compatibility with all IO models: non-blocking, thready, async