graviola

fast, high-assurance
cryptography for Rust

about me

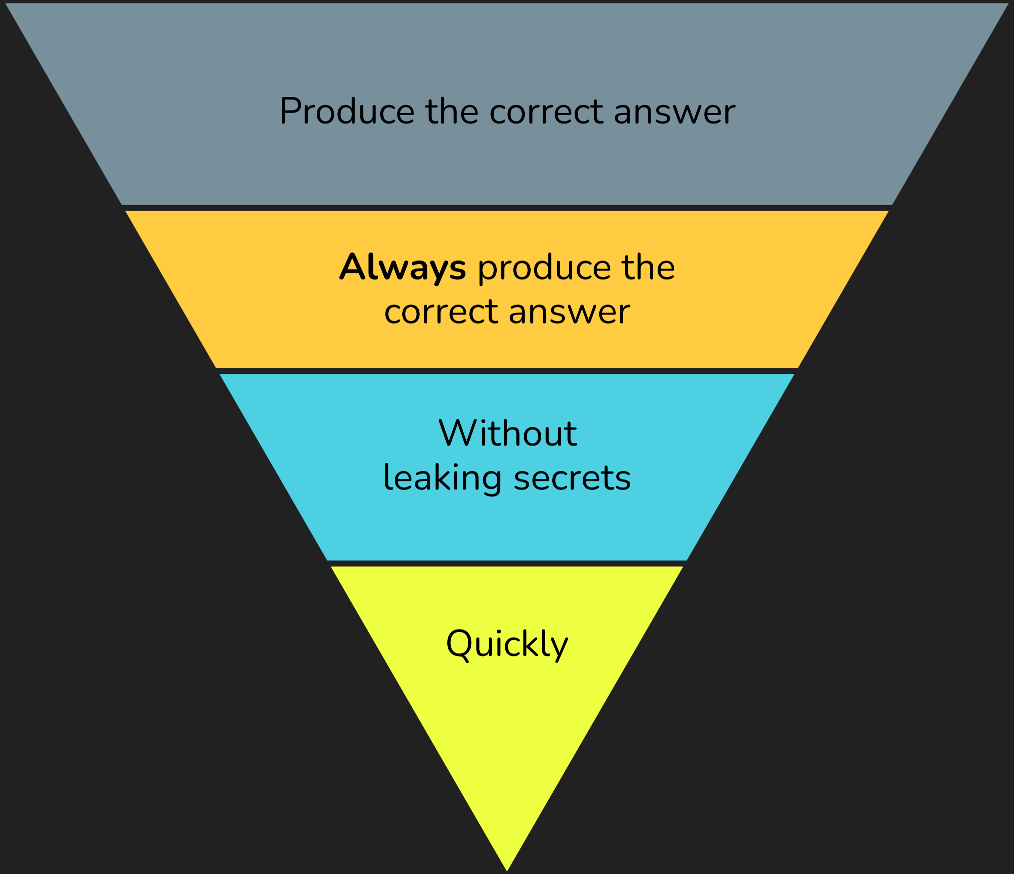
original author and now co-maintainer of rustls

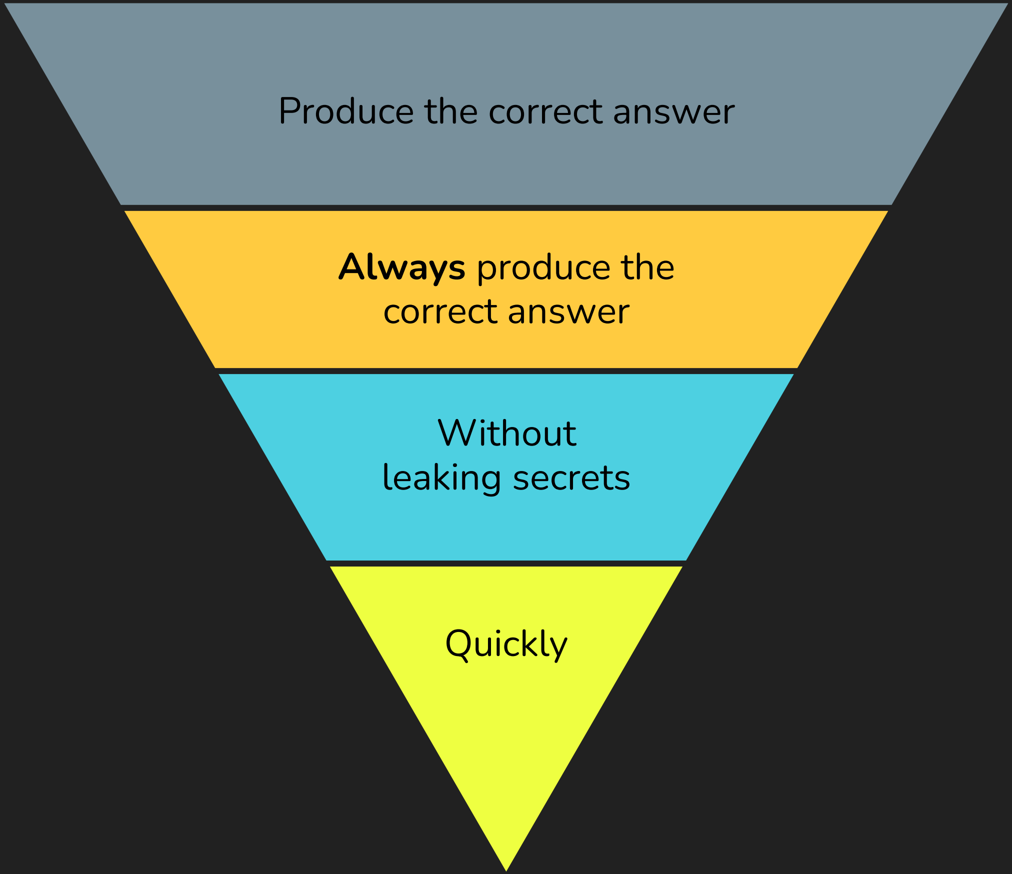
writing rust since october 2015

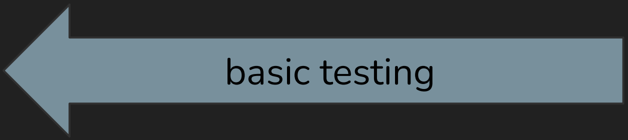
this talk

- what do we want from cryptography code anyway?
  - about side-channels
  - why optimising compilers are bad for cryptography code
- s2n-bignum: formally verified assembly for low level crypto operations
  - using assembly from rust
- graviola
  - features & limitations
  - performance
  - structure
  - some nice details

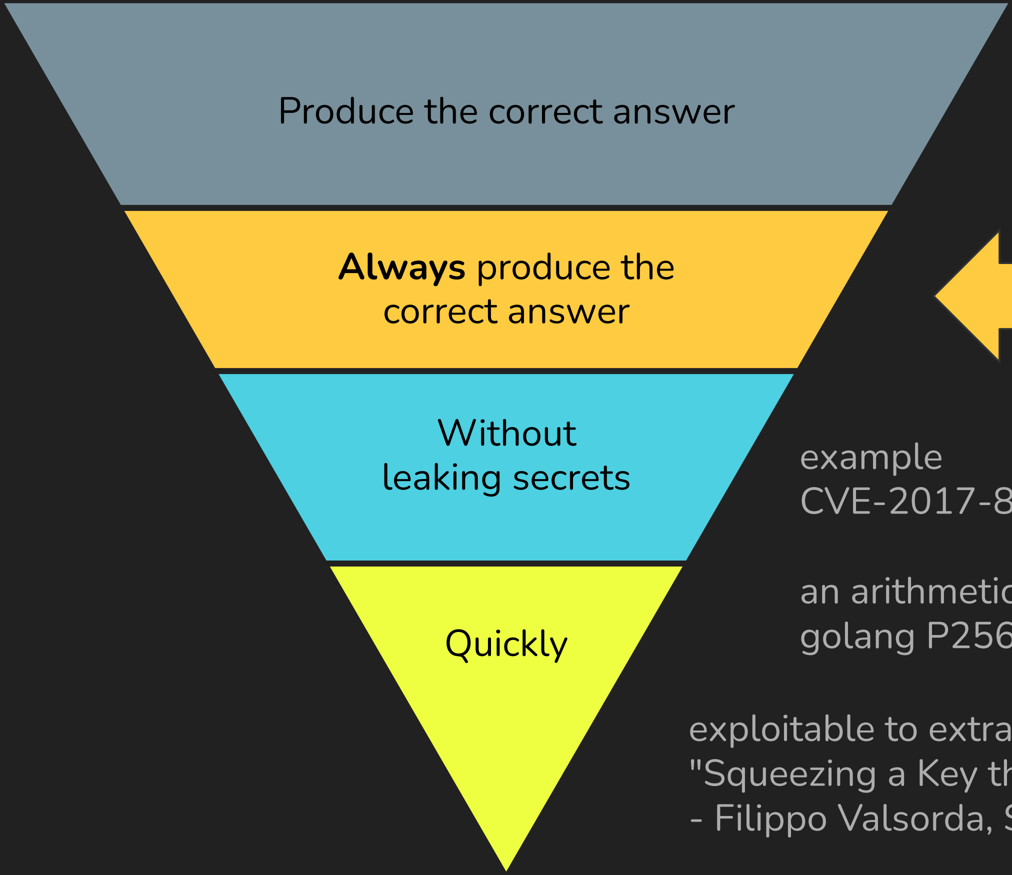
what do we want from cryptography code anyway?

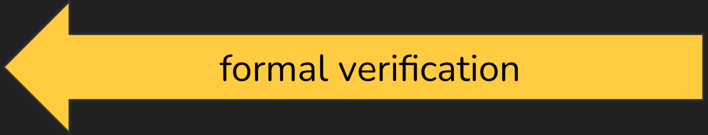

Produce the correct answer
Always produce the correct answer
Without leaking secrets
Quickly





cryptography code is
(at its core) deterministic and
embarrassingly easy to test



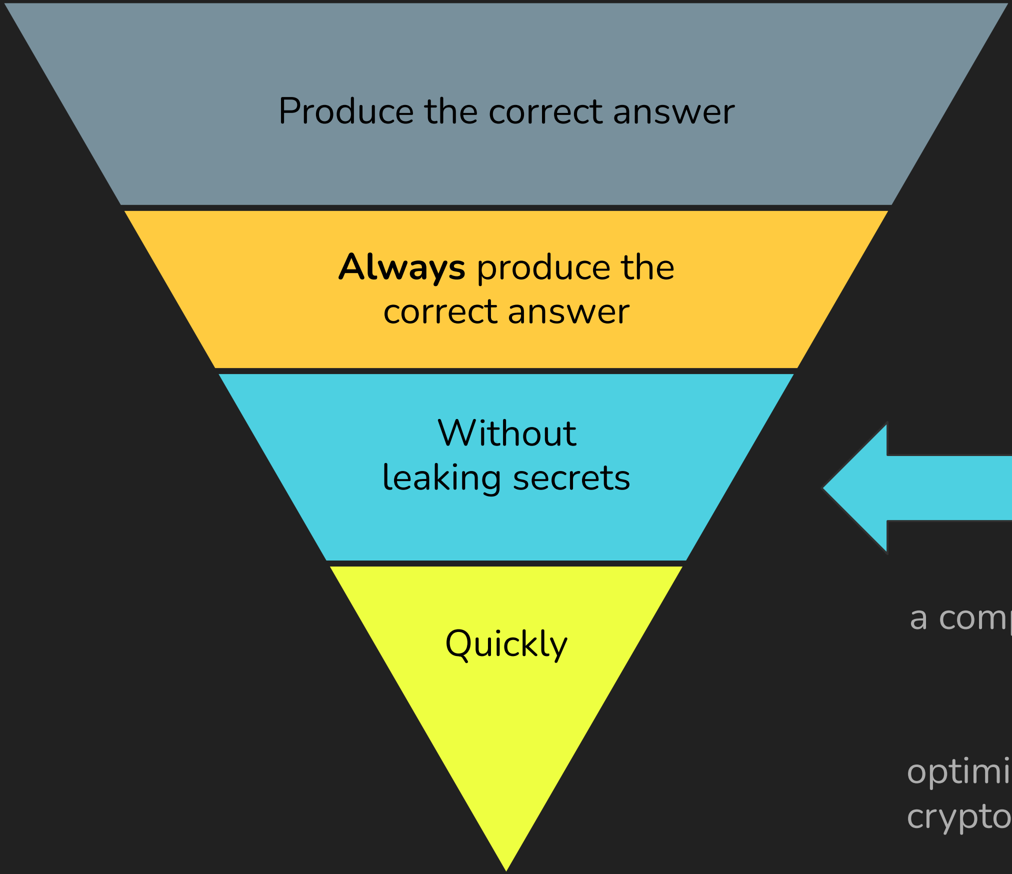


normal software testing approaches are not enough here: input space is too large to find errors at random.

example CVE-2017-8932:

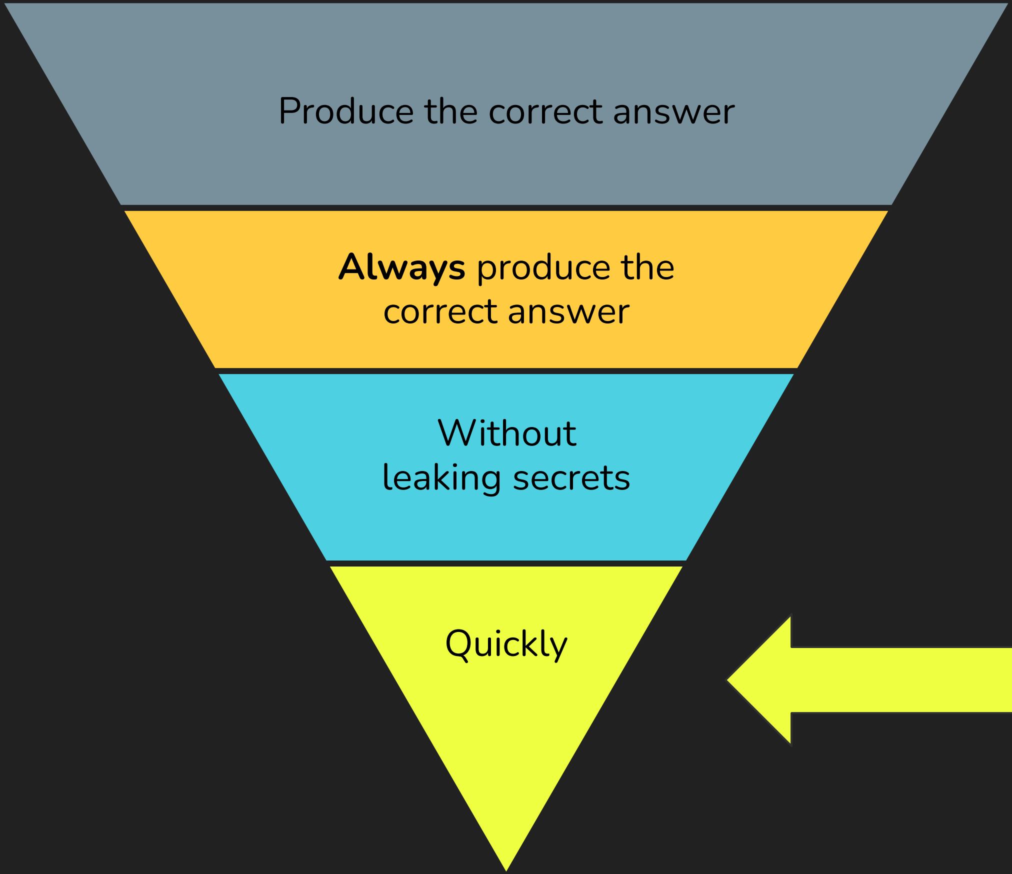
an arithmetic error in golang P256

exploitable to extract private key data: see "Squeezing a Key through a Carry Bit"
- Filippo Valsorda, Sean Devlin - Blackhat US 2018



a complex and current problem

optimising compilers are bad for cryptography code



measure - understand - improve cycle

lots of literature on strategies to make things fast

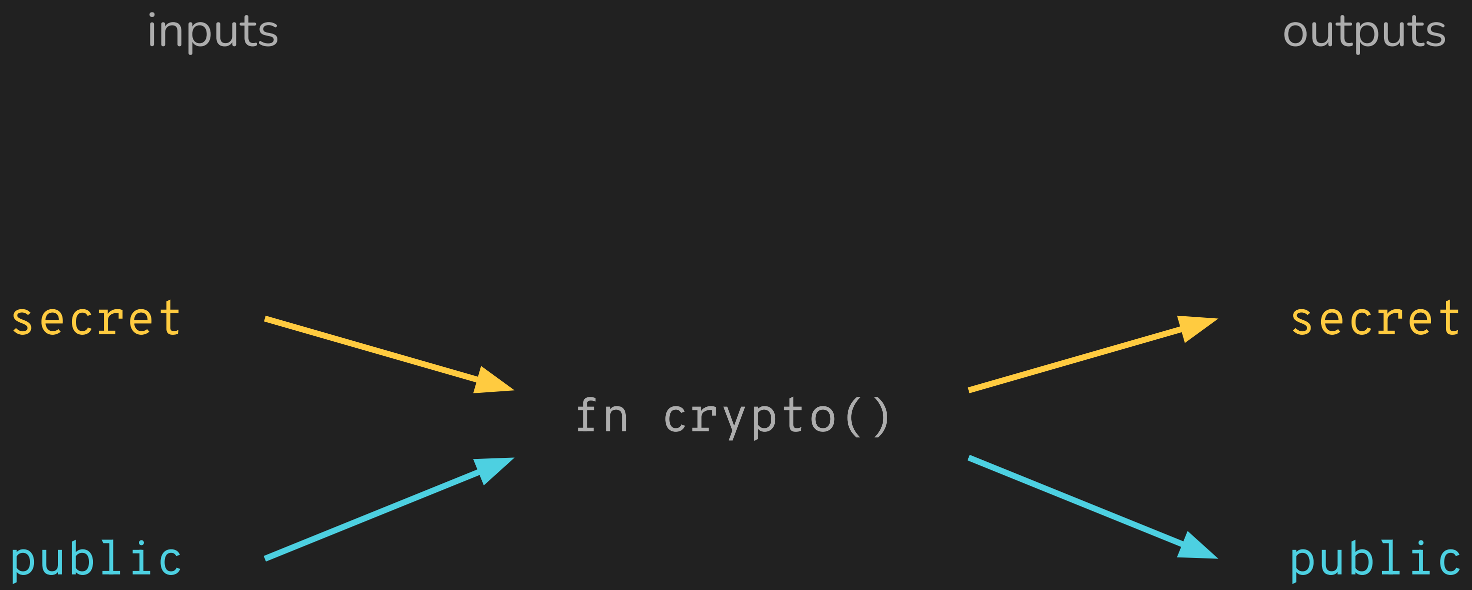
lots of open-source, fast cryptography to learn from

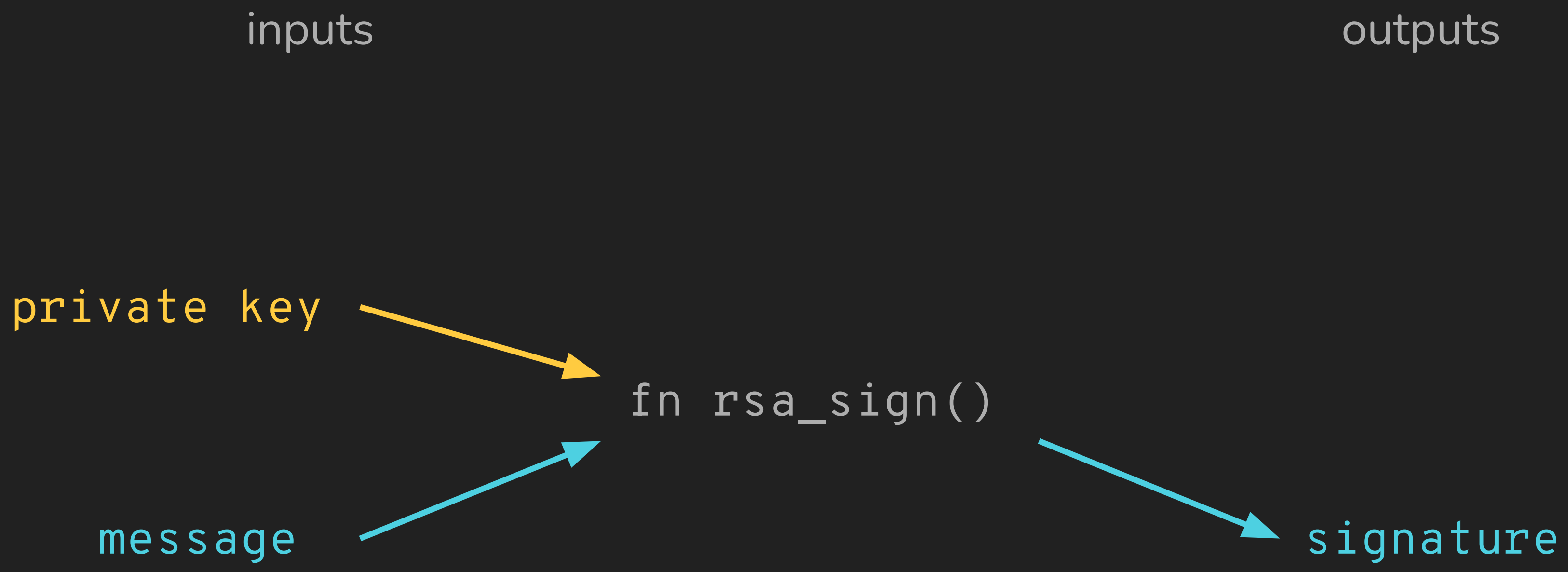
matter of time, effort, and "mechanical sympathy"

performance analysis

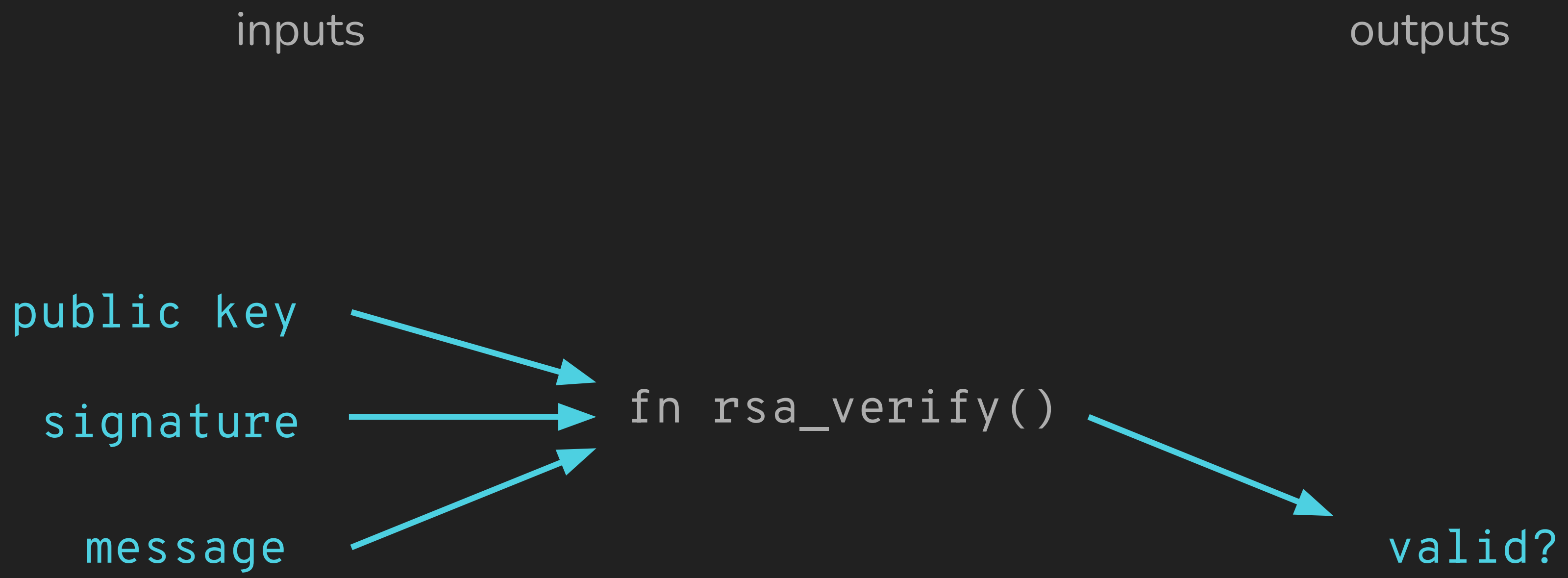
about side-channels

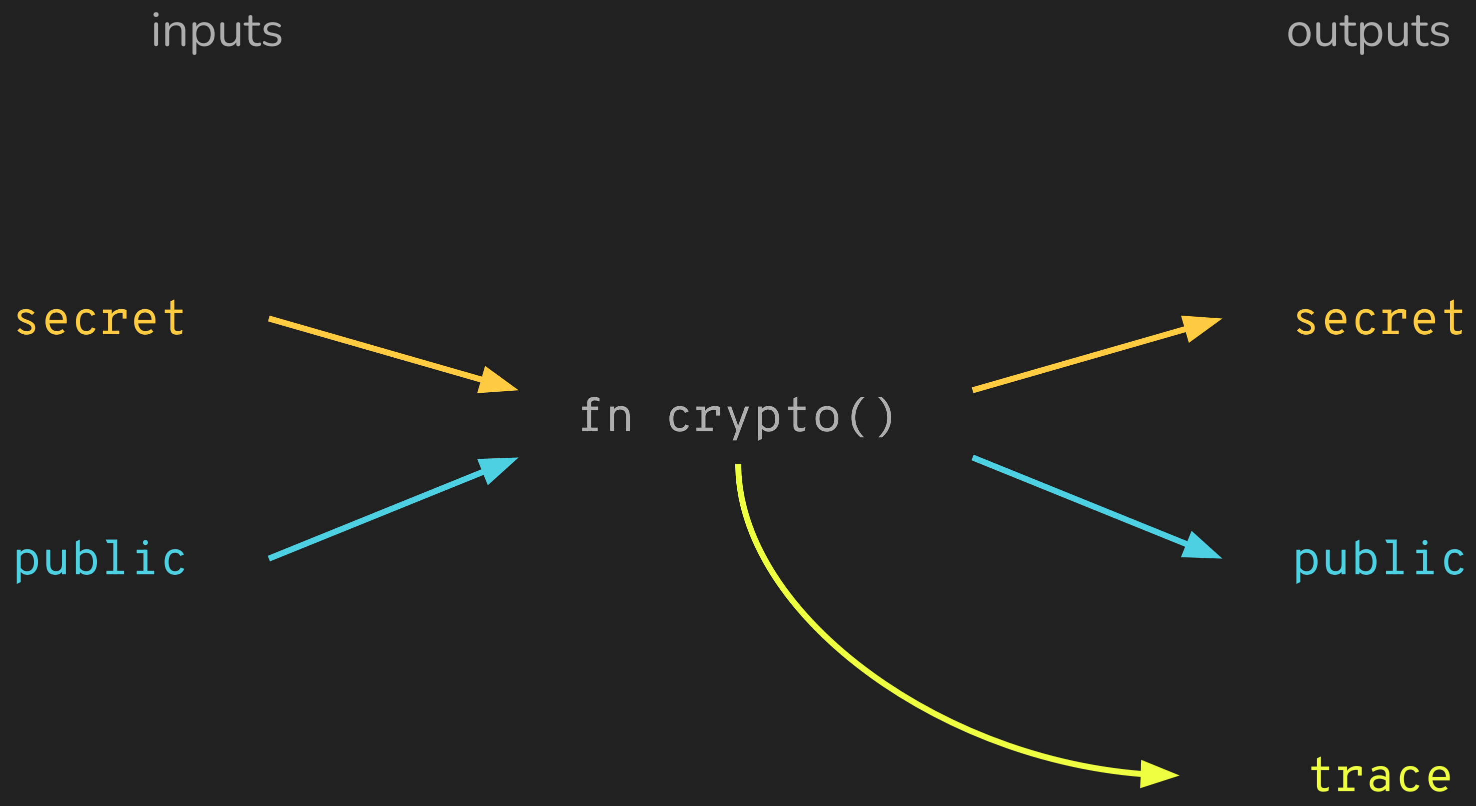
fn crypto()


inputs
outputs
secret
public
fn crypto()
secret
public



inputs
outputs
private key
fn rsa_sign()
message
signature


inputs
outputs
public key
signature
message
fn rsa_verify()
valid?


inputs
outputs
secret
public
fn crypto()
secret
public
trace

trace

instruction trace

every instruction executed

```
mov     r9, r8
sar     r9, 0x3f
xor     r8, r9
sub     r8, r9
mov     r11, r10
sar     r11, 0x3f
xor     r10, r11
sub     r10, r11
mov     r13, r12
sar     r13, 0x3f
…
```

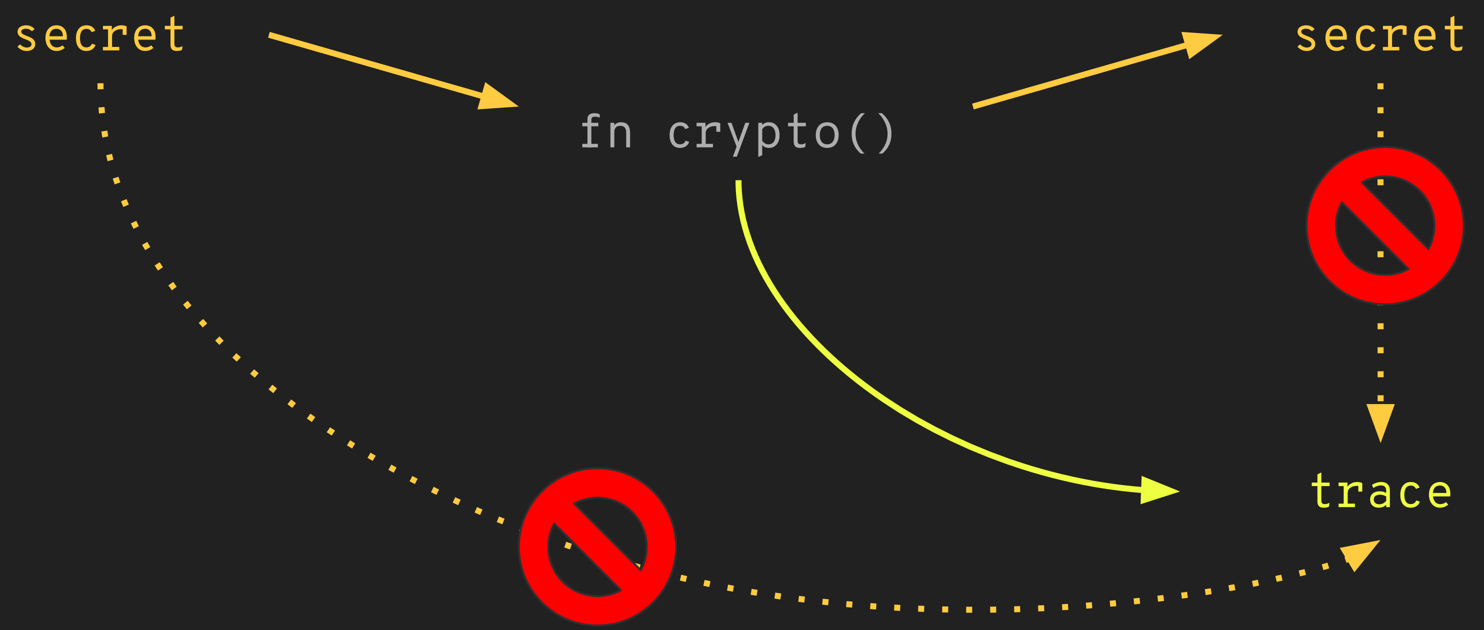
+

memory trace

every memory access

```
read 32b@0x7fefefa006c0
write 32b@0x7fefefa006c0
…
```

nb. register and memory contents **not** included goal 1: trace has no dependency on secrets



goal 1: trace has no dependency on secrets

instruction trace dependent on secret?
→ timing, branch prediction, EM, simple power side-channels, …

memory trace dependent on secret?
→ cache side-channel, …

goal 2: instruction trace must contain no instructions with data-dependent timing operating on secret data

otherwise, timing or EM side channel, or simple power analysis

- → avoid those instructions (eg, division*)
- → on ARM: ensure functions with any secret inputs or outputs set "Data Independent Timing" (DIT) bit

*: https://kyberslash.cr.yp.to/

optimising compilers are bad for cryptography

optimising compilers are bad for cryptography

(this includes rustc - and every competitive c and c++ compiler) why?

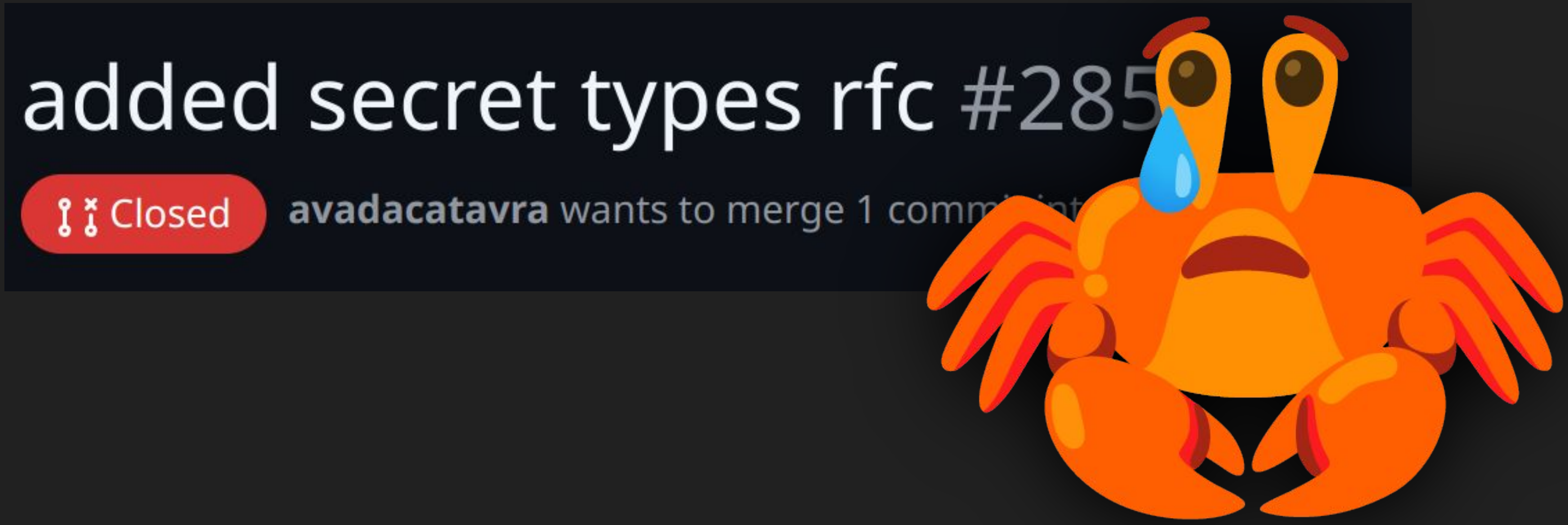
we can't tell the optimiser about our side channel goals

lots of broken workarounds abound

there is a design to improve this, for rustc

added secret types rfc #2859

there is a design to improve this, for rustc



there is a design to improve this, for rustc

> All cryptographic code written in higher-level languages than assembly makes an effort to try to use code that compilers don't screw up and then essentially hope for the best.

\- Peter Schwabe

CVE-2024-37880 - clang 18 inserts branch side channel into Kyber

RUSTSEC-2024-0344 - rustc inserts branch side channel into curve25519-dalek

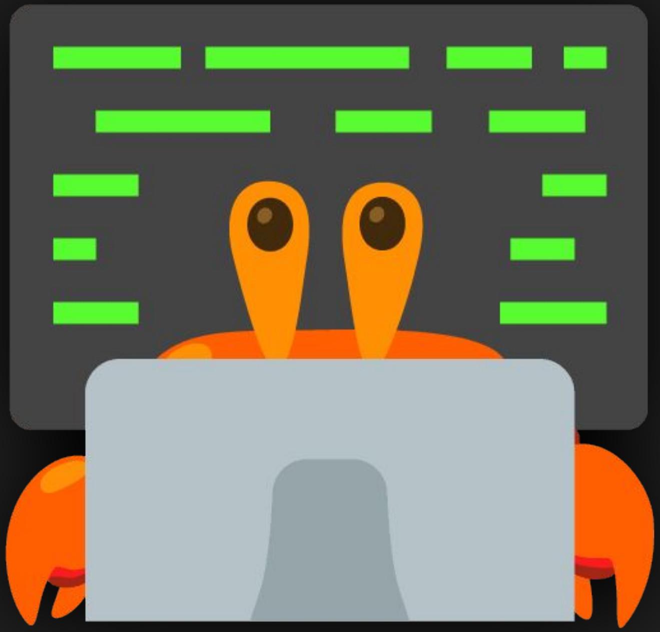
enter s2n-bignum

https://github.com/awslabs/s2n-bignum/

formally verified cryptography routines in aarch64 and x86_64 assembly

"formally verified" - each function proven to implement exactly the specified mathematical operation

see their readme for details on side-channel safety



using assembly from rust

Using assembly from rust

options:

1) "traditional"

(build time)

unfortunately:

- getting an assembler, c preprocessor, etc is v. annoying on some platforms
- prevents inlining
- significant build-time cost and complexity
- function exit/entry ABI is platform-specific

Using assembly from rust

options:

1) ~~"traditional"~~

2) transpile assembly to rust (containing inline assembly)

(development time)

fortunately:

- build just requires rustc
- inlining works
- ~zero build time cost and complexity
- rustc handles symbol naming & entry/exit ABI

```c
#define p rdi
#define z rsi
```

```rust
macro_rules! p {
    () => {
        "rdi"
    };
}
macro_rules! z {
    () => {
        "rsi"
    };
}
```

```c
#define p rdi
#define z rsi
```

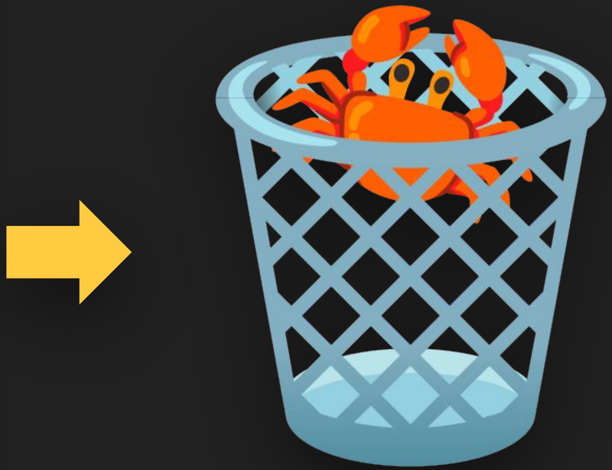
this leads to many macros in the crate: ~1400 in total

```rust
macro_rules! p {
    () => {
        "rdi"
    };
}
macro_rules! z {
    () => {
        "rsi"
    };
}
```

```
#define mulpadd(high,low,m)             \
        mulx    rcx, rax, m;            \
        adcx    low, rax;               \
        adox    high, rcx
```

```
macro_rules! mulpadd {
    ($high:expr, $low:expr, $m:expr) => { Q!(
        "mulx rcx, rax, " $m ";\n"
        "adcx " $low ", rax;\n"
        "adox " $high ", rcx"
    )}
}
```

```
#if WINDOWS_ABI
        push    rdi
        push    rsi
        mov     rdi, rcx
        mov     rsi, rdx
        mov     rdx, r8
#endif
```



```
// Zero the main index counter for both branches

        xor     i, i

// First clamp the two input sizes m := min(p,m) and n := min(p,n) since
// we'll never need words past the p'th. Can now assume m <= p and n <= p.
// Then compare the modified m and n and branch accordingly

        cmp     p, m
        cmovc   m, p
        cmp     p, n
        cmovc   n, p
        cmp     m, n
        jc      ylonger
```

```
// Zero the main index counter for both branches

Q!("    xor            " i!() ", " i!()),

// First clamp the two input sizes m := min(p,m) and n := min(p,n) since
// we'll never need words past the p'th. Can now assume m <= p and n <= p.
// Then compare the modified m and n and branch accordingly

Q!("    cmp            " p!() ", " m!()),
Q!("    cmovc          " m!() ", " p!()),
Q!("    cmp            " p!() ", " n!()),
Q!("    cmovc          " n!() ", " p!()),
Q!("    cmp            " m!() ", " n!()),
Q!("    jc             " Label!("ylonger", 2, After)),
```

```rust
pub(crate) fn bignum_add(z: &mut [u64], x: &[u64], y: &[u64]) {
    // SAFETY: inline assembly. see [crate::low::inline_assembly_safety] for safety info.
    unsafe {
        core::arch::asm!(
        // ...
        inout("rdi") z.len() => _,
        inout("rsi") z.as_mut_ptr() => _,
        inout("rdx") x.len() => _,
        inout("rcx") x.as_ptr() => _,
        inout("r8") y.len() => _,
        inout("r9") y.as_ptr() => _,
        // clobbers
        out("r10") _,
        out("rax") _,
        )
    };
}
```

non-automated elements

about graviola

it's a fruit, but that's not important right now

about graviola (the crate)

goals:

- easy and fast to build
- for use with rustls - commonly-used cryptography for TLS
- competitive performance
- under a non-weird license

about graviola (the crate)

achievements:

- easy and fast to build:
  - just rustc. no build.rs, no proc-macros
  - ~1 second build time
  - two dependencies: cfg-if & getrandom
- licensed under ISC + Apache2.0 + MIT-0

features

Public key signatures

- RSA-PSS signing & verification
- RSA-PKCS#1 signing & verification
- ECDSA on P256 w/ SHA2
- ECDSA on P384 w/ SHA2

Hashing

- SHA256, SHA384 & SHA512
- HMAC & HMAC-DRBG

Key exchange

- X25519
- P256
- P384

AEADs

- AES-GCM
- chacha20-poly1305

features

Public key signatures

- RSA-PSS signing & verification
- RSA-PKCS#1 signing & verification
- ECDSA on P256 w/ SHA2
- ECDSA on P384 w/ SHA2

Key exchange

- X25519
- P256
- P384

constructed atop s2n-bignum

Hashing

- SHA256, SHA384 & SHA512
- HMAC & HMAC-DRBG

AEADs

- AES-GCM
- chacha20-poly1305

new rust code, using intrinsics

limitations

- x86_64-v3 and aarch64 CPU architectures only
  - x86_64: *most* CPUs since 2013-2014
  - ARM aarch64: all apple M, ~all server-grade ARMs, RPi 5 or later
- widely used cryptography only

how to use it

integration with rustls is in its own crate: `rustls-graviola`

```
rustls-graviola v0.2.0
├── graviola v0.2.0
│   ├── cfg-if v1.0.0
│   └── getrandom v0.3.1
│       ├── cfg-if v1.0.0
│       └── libc v0.2.168
└── rustls v0.23.19
    └── …
```

```rust
rustls_graviola::default_provider()
    .install_default()
    .unwrap();
```

performance – see https://jbp.io/graviola/

x86_64

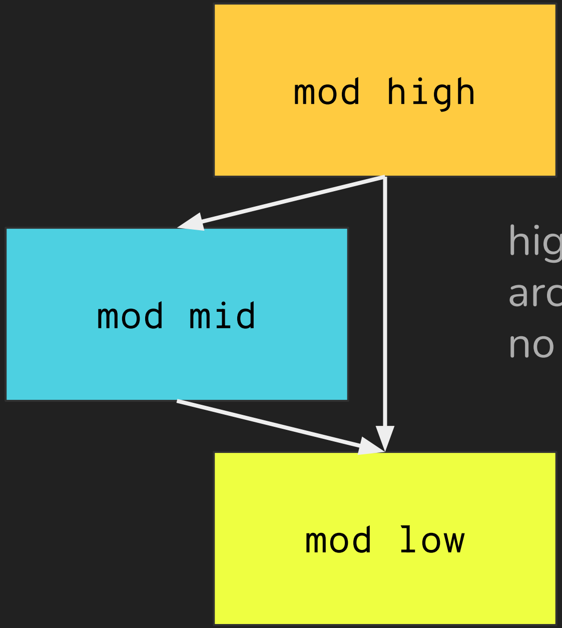
Signing

RSA2048 signing	**aws-lc-rs** ⓘ 5,377.8 sigs/sec	**ring** 2,446.5 sigs/sec	**graviola** 2,337.8 sigs/sec	**golang** 1,358.4 sigs/sec	**rustcrypto** 884.28 sigs/sec
ECDSA-P256 signing	**graviola** 91,486 sigs/sec	**ring** 81,536 sigs/sec	**aws-lc-rs** 78,292 sigs/sec	**golang** 60,809 sigs/sec	**rustcrypto** 8,607.2 sigs/sec
ECDSA-P384 signing	**aws-lc-rs** 15,718 sigs/sec	**graviola** 9,153.4 sigs/sec	**golang** 6,798 sigs/sec	**ring** 3,297.5 sigs/sec	**rustcrypto** 2,172 sigs/sec

performance – see https://jbp.io/graviola/

	aarch64 (ARM)	x86_64 (Intel)
RSA2048 signing	🥉 3rd	🥉 3rd
ECDSA-P256 signing	🥇 1st	🥇 1st
ECDSA-P384 signing	🥈 2nd	🥈 2nd
RSA2048 signature verification	🥉 3rd	🥇 1st
ECDSA-P256 signature verification	🥇 1st	🥈 2nd
ECDSA-P384 signature verification	🥈 2nd	🥈 2nd
X25519 key agreement	🥇 1st	🥇 1st
P256 key agreement	🥇 1st	🥈 2nd
P384 key agreement	🥈 2nd	🥈 2nd
AES256-GCM encryption (8KB wide)	🥉 3rd	🥉 3rd

graviola internal structure

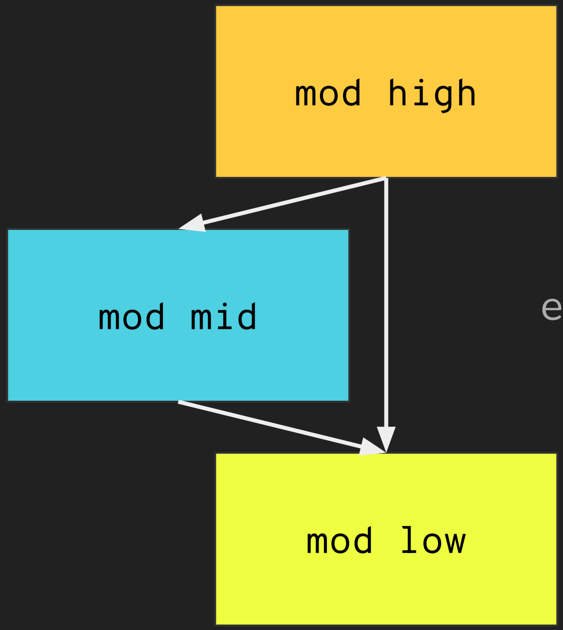


parsing and encoding
architecture agnostic
no unsafe

high-level arithmetic
architecture agnostic
no unsafe

low-level primitives
private
perhaps architecture-specific
unsafe allowed

graviola internal structure



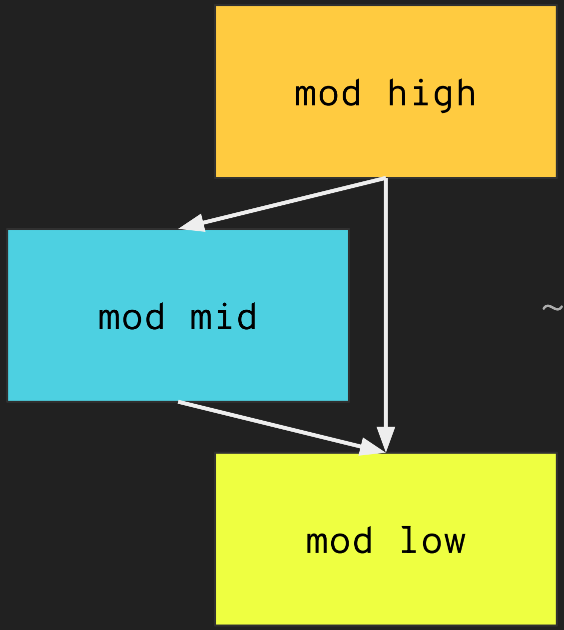
eg, ASN.1 decoding an RSA key

eg, RSA private operation

eg, multiplying large numbers

(mostly: safe functions backed by s2n-bignum)

graviola internal structure



~3900 lines

~3900 lines

~67,000 lines
of which
~61,000 lines from s2n-bignum
(even split between aarch64 and x86_64)

more macro abuse

macro-based ASN.1 decoder:

```
RSAPublicKey ::= SEQUENCE {
    modulus            INTEGER,    -- n
    publicExponent     INTEGER  }  -- e
```

```
asn1_struct! {
    RSAPublicKey ::= SEQUENCE {
        modulus              INTEGER,
        publicExponent       INTEGER
    }
}

→

struct RSAPublicKey { … }
RSAPublicKey::from_bytes(bytes)
```

more macro abuse

macro-based ASN.1 decoder:

```
secp384r1 OBJECT IDENTIFIER ::= {
    iso(1) identified-organization(3)
    certicom(132) curve(0) 34
}
```

```
asn1_oid! {
    secp384r1 OBJECT IDENTIFIER ::= {
        iso(1) identified_organization(3)
        certicom(132) curve(0) 34
    }
}
```

→

```
pub(crate) static secp384r1:
crate::high::asn1::ObjectId = …;
```

designating functions as secret/public

`Entry` type:

every pub function entrypoint starts with

```
let _entry = Entry::new_secret();
```

or

```
let _entry = Entry::new_public();
```

"secret" functions:

- set ARM "Data Independent Timing" (DIT) flag on entry (if needed), reset on return
- clear vector registers on return
- *future*: stack zeroisation
- *future*: scalar register zeroisation

parting words

using assembly for cryptography code avoids side-channel hazards in optimising compilers

and usually gets good performance too!
but this alone doesn't give you side-channel free results!

stand-alone functions in assembly are easy* to use from "pure" rust, even if they use the c preprocessor

** terms and conditions apply*

graviola is quick to build, has competitive performance, and is ready to use with rustls

for supported architectures

thanks!

Repo: https://github.com/ctz/graviola

BlueSky: https://bsky.app/profile/jbp.io

Mail: jbp@jbp.io

Slides: https://github.com/ctz/talks