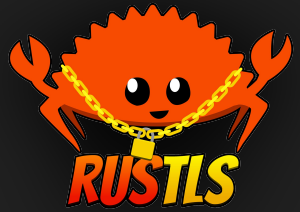
graviola

fast, high-assurance cryptography for Rust

about me



original author and now co-maintainer of rustls

writing rust since october 2015

but first, some unfinished business...

CVE-2024-5535: `SSL_select_next_proto` buffer overread

Affected all versions of OpenSSL 1.0.x, 1.1.x, 3.x, plus BoringSSL.

Bug introduced in 2011, fixed in May 2024.

```
require('tls').connect({port: 443, NPNProtocols: new Uint8Array()}, function(c) {})
```

Not exploitable in recent times (affects a TLS protocol feature that is now largely unused – never standardised and abandoned in 2012).

But documented in wild by okhttp developer on Android in 2014. (Android not vulnerable from 2014 on.)

https://jbp.io/2024/06/27/cve-2024-5535-openssl-memory-safety.html
https://openssl-library.org/news/vulnerabilities/#CVE-2024-5535

this talk

does rust need more cryptography options?

why optimising compilers are bad for cryptography code

- s2n-bignum: formally verified assembly for low level crypto operations
- using assembly from rust

graviola

- goals
- features
- limitations
- performance

does rust need more cryptography options?

Portability Build simplicity Performance Assurance

caveat

not a scientific assessment

just my opinion

	Portability	Build simplicity	Performance	Assurance
aws-lc-rs Apache-2.0, ISC, OpenSSL				

	Portability	Build simplicity	Performance	Assurance
aws-lc-rs Apache-2.0, ISC, OpenSSL				
ring Apache-2.0, ISC ~~OpenSSL~~				

	Portability	Build simplicity	Performance	Assurance
aws-lc-rs Apache-2.0, ISC, OpenSSL				
ring Apache-2.0, ISC ~~OpenSSL~~				
Rust Crypto Apache-2.0, MIT				
Dalek BSD-3-Clause				

	Portability	Build simplicity	Performance	Assurance
aws-lc-rs Apache-2.0, ISC, OpenSSL				
ring Apache-2.0, ISC ~~OpenSSL~~				
Rust Crypto Apache-2.0, MIT				
Dalek BSD-3-Clause				
Graviola Apache-2.0, MIT, ISC				

optimising compilers are kryptonite for cryptography

optimising compilers are kryptonite for cryptography

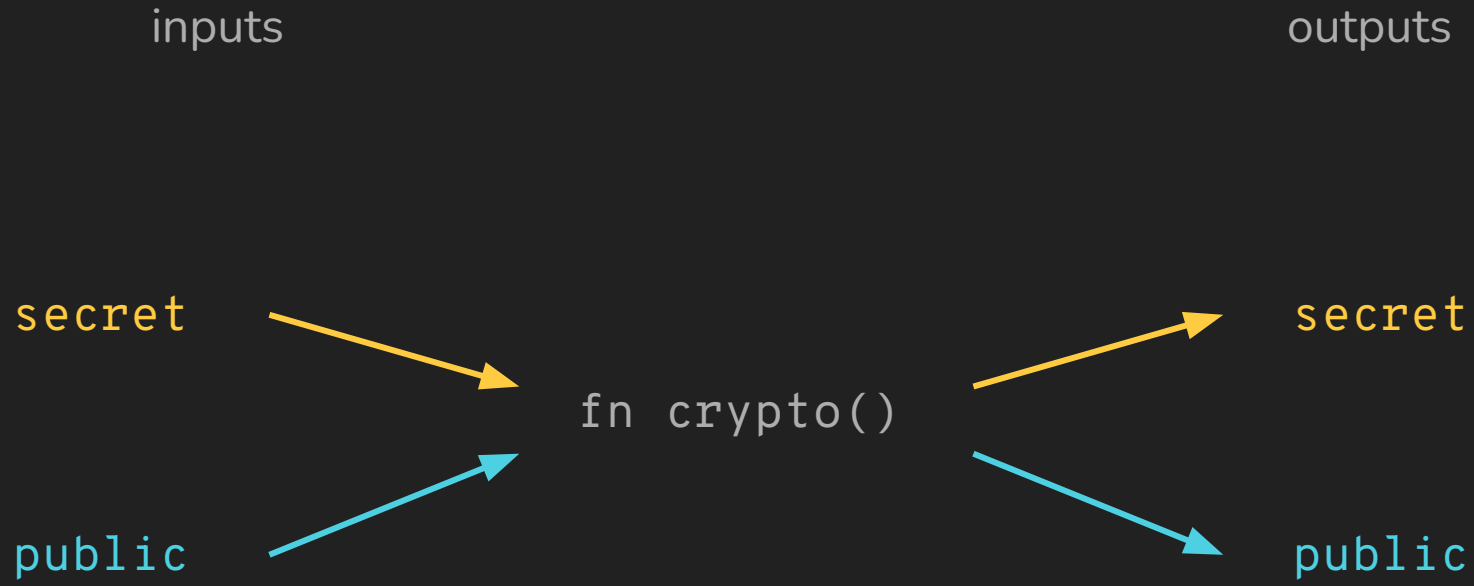
(this includes rustc - and every competitive c and c++ compiler)

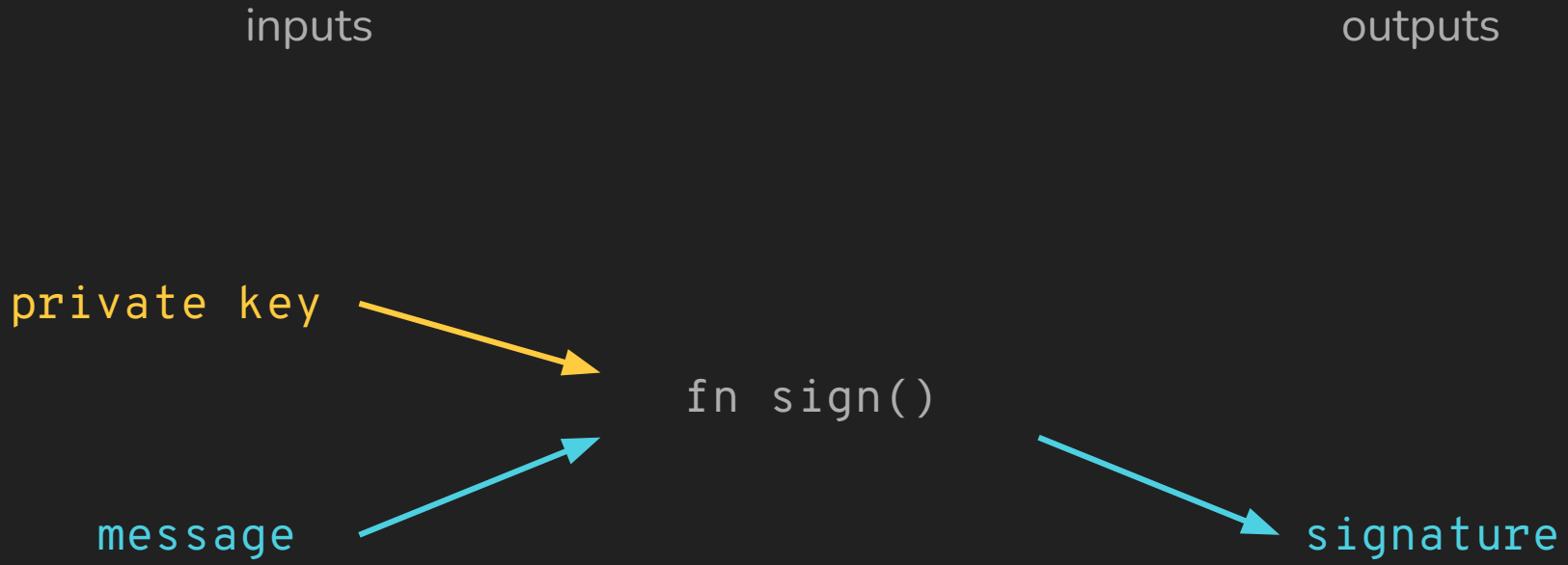
why?

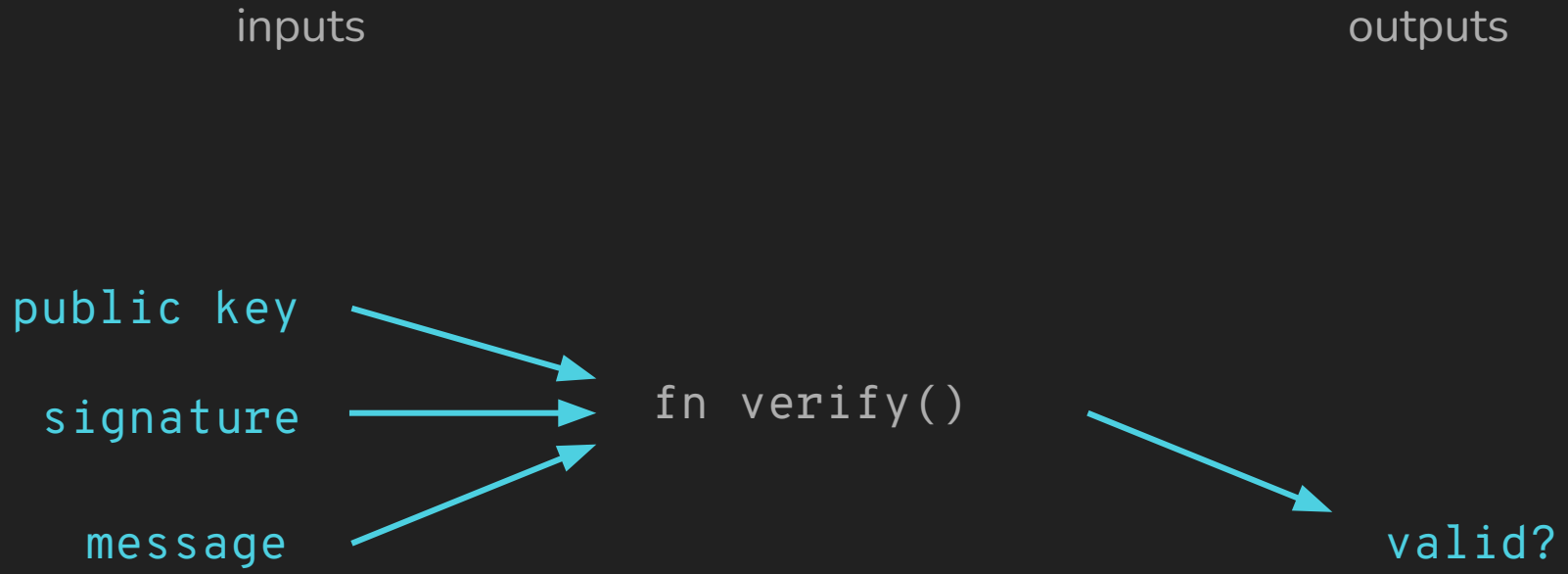
we can't tell the optimiser about
our side-channel goals

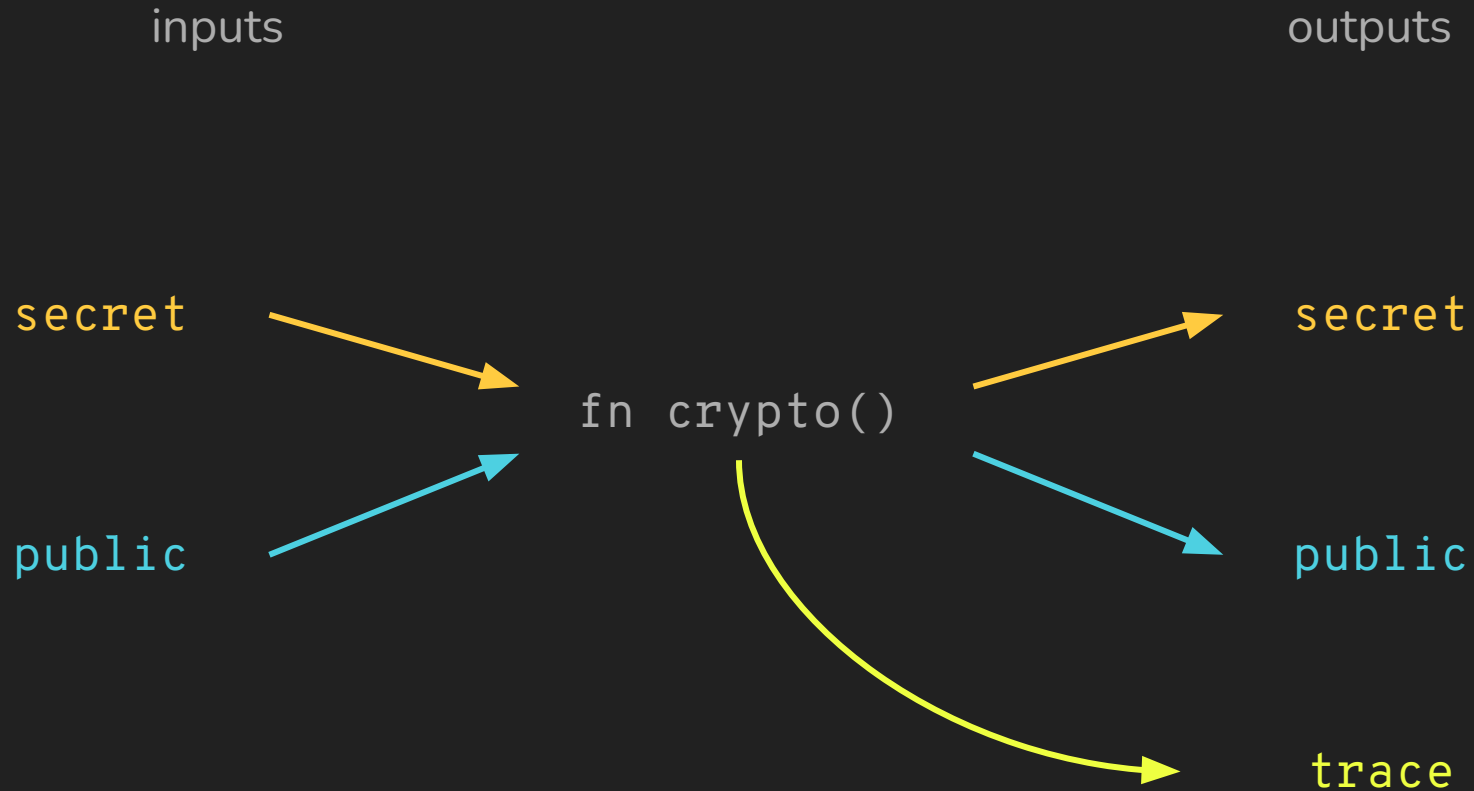
about
side-channels

fn crypto()


inputs
outputs
secret
public
fn crypto()
secret
public


inputs
outputs
private key
message
fn sign()
signature


inputs
outputs
public key
signature
message
fn 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
…
```

+

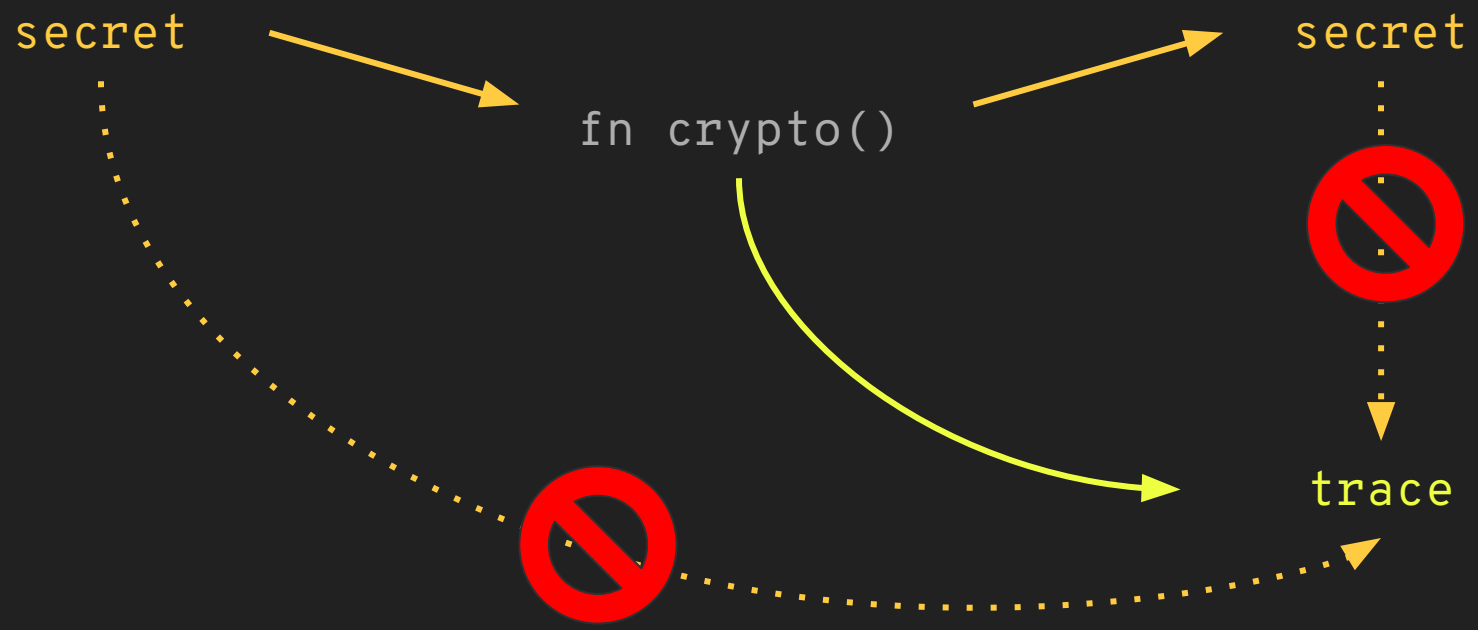
memory trace

every memory access

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

nb. register and memory contents **not** included

goal 1: trace has no dependency on secrets

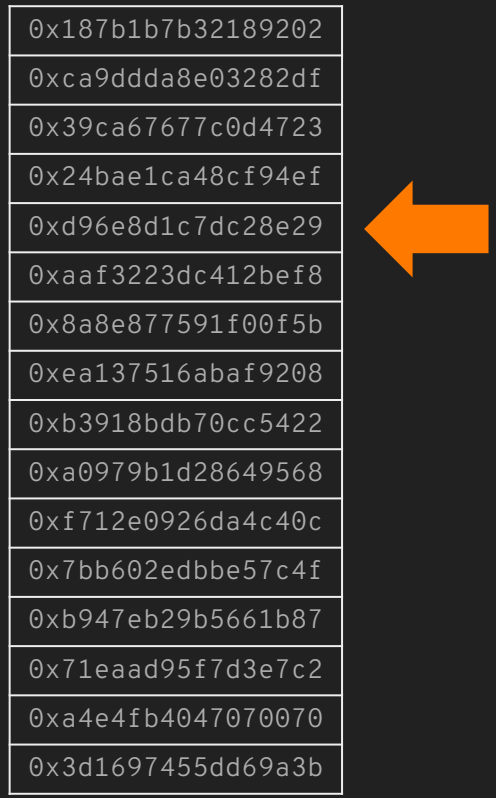


goal 1: trace has no dependency on secrets

- → no branches depending on secret data
- → no memory accesses depending on secret data

  (but can do "table selection" accesses)

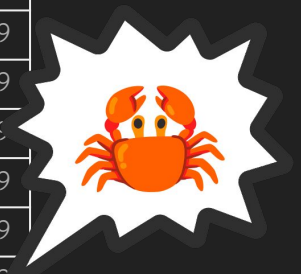
data


0x187b1b7b32189202
0xca9ddda8e03282df
0x39ca67677c0d4723
0x24bae1ca48cf94ef
0xd96e8d1c7dc28e29
0xaaf3223dc412bef8
0x8a8e877591f00f5b
0xea137516abaf9208
0xb3918bdb70cc5422
0xa0979b1d28649568
0xf712e0926da4c40c
0x7bb602edbbe57c4f
0xb947eb29b5661b87
0x71eaad95f7d3e7c2
0xa4e4fb4047070070
0x3d1697455dd69a3b

data	index	wanted
0x187b1b7b32189202	0	4
0xca9ddda8e03282df	1	4
0x39ca67677c0d4723	2	4
0x24bae1ca48cf94ef	3	4
0xd96e8d1c7dc28e29	4	4
0xaaf3223dc412bef8	5	4
0x8a8e877591f00f5b	6	4
0xea137516abaf9208	7	4
0xb3918bdb70cc5422	8	4
0xa0979b1d28649568	9	4
0xf712e0926da4c40c	10	4
0x7bb602edbbe57c4f	11	4
0xb947eb29b5661b87	12	4
0x71eaad95f7d3e7c2	13	4
0xa4e4fb4047070070	14	4
0x3d1697455dd69a3b	15	4

data	index	wanted	mask
0x187b1b7b32189202	0	4	0x0000000000000000
0xca9ddda8e03282df	1	4	0x0000000000000000
0x39ca67677c0d4723	2	4	0x0000000000000000
0x24bae1ca48cf94ef	3	4	0x0000000000000000
0xd96e8d1c7dc28e29	4	4	0xFFFFFFFFFFFFFFFF
0xaaf3223dc412bef8	5	4	0x0000000000000000
0x8a8e877591f00f5b	6	4	0x0000000000000000
0xea137516abaf9208	7	4	0x0000000000000000
0xb3918bdb70cc5422	8	4	0x0000000000000000
0xa0979b1d28649568	9	4	0x0000000000000000
0xf712e0926da4c40c	10	4	0x0000000000000000
0x7bb602edbbe57c4f	11	4	0x0000000000000000
0xb947eb29b5661b87	12	4	0x0000000000000000
0x71eaad95f7d3e7c2	13	4	0x0000000000000000
0xa4e4fb4047070070	14	4	0x0000000000000000
0x3d1697455dd69a3b	15	4	0x0000000000000000

data		mask		accumulated result
0x187b1b7b32189202	&	0x0000000000000000	^=	0x0000000000000000
0xca9ddda8e03282df	&	0x0000000000000000	^=	0x0000000000000000
0x39ca67677c0d4723	&	0x0000000000000000	^=	0x0000000000000000
0x24bae1ca48cf94ef	&	0x0000000000000000	^=	0x0000000000000000
0xd96e8d1c7dc28e29	&	0xFFFFFFFFFFFFFFFF	^=	0xd96e8d1c7dc28e29
0xaaf3223dc412bef8	&	0x0000000000000000	^=	0xd96e8d1c7dc28e29
0x8a8e877591f00f5b	&	0x0000000000000000	^=	0xd96e8d1c7dc28e29
0xea137516abaf9208	&	0x0000000000000000	^=	0xd96e8d1c7dc28e29
0xb3918bdb70cc5422	&	0x0000000000000000	^=	0xd96e8d1c7dc28e29
0xa0979b1d28649568	&	0x0000000000000000	^=	0xd96e8d1c7dc28e29
0xf712e0926da4c40c	&	0x0000000000000000	^=	0xd96e8d1c7dc28e29
0x7bb602edbbe57c4f	&	0x0000000000000000	^=	0xd96e8d1c7dc28e29
0xb947eb29b5661b87	&	0x0000000000000000	^=	0xd96e8d1c7dc28e2
0x71eaad95f7d3e7c2	&	0x0000000000000000	^=	0xd96e8d1c7dc28e29
0xa4e4fb4047070070	&	0x0000000000000000	^=	0xd96e8d1c7dc28e29
0x3d1697455dd69a3b	&	0x0000000000000000	^=	0xd96e8d1c7dc28e2



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

- → 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/

summary:

1. think precisely about secret vs public data
2. extreme care when processing secret data

(micro-architectural problems still happen, but are not under the control of software authors)

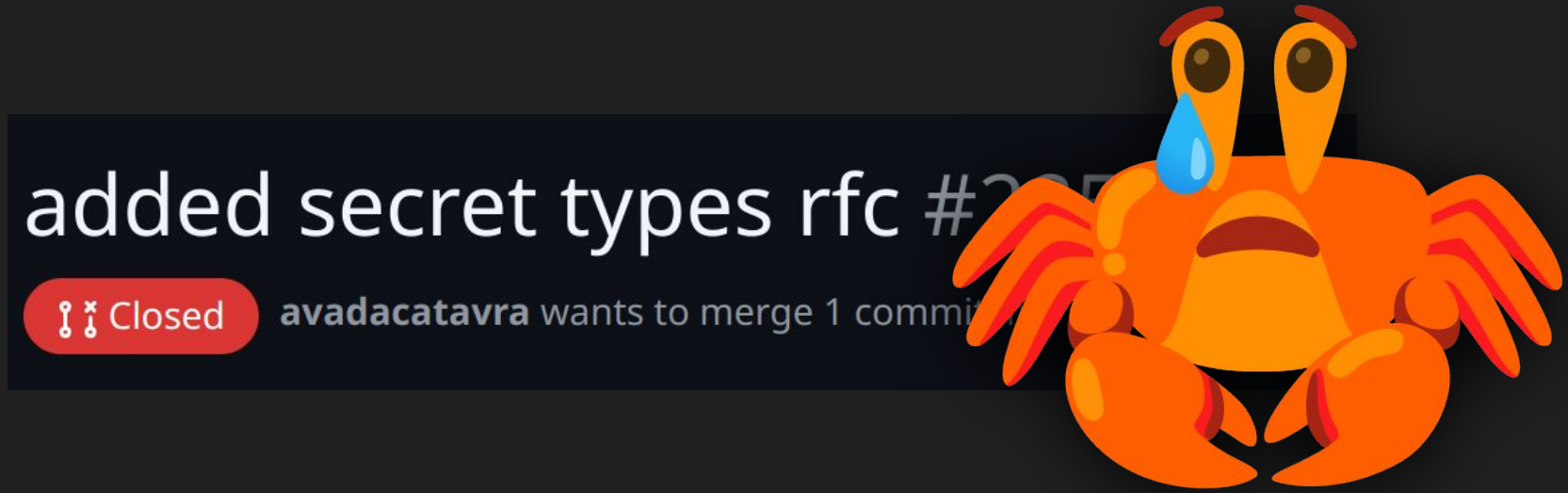
back to optimizing compilers...

there is a design to improve this, for rustc

```
added secret types rfc #2859
```

back to optimizing compilers...

there is a design to improve this, for rustc



back to optimizing compilers...

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

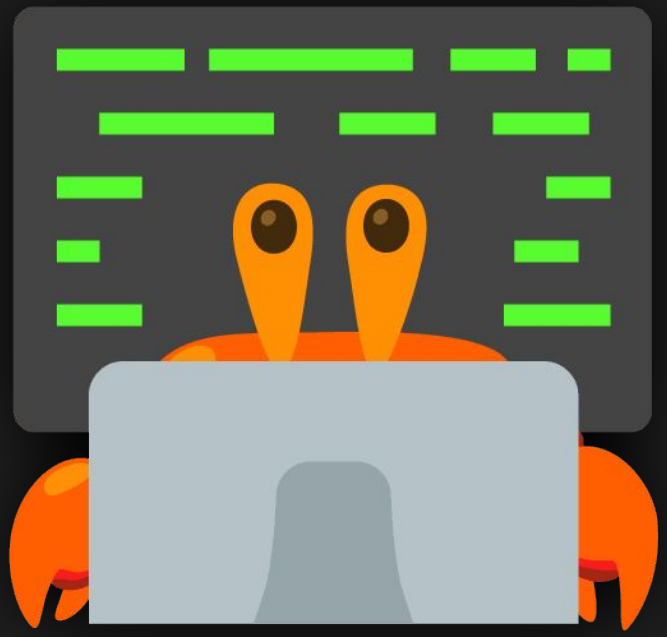
enter s2n-bignum

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

formally verified cryptography routines in aarch64 and x86_64 assembly

addresses entire classes of correctness bugs such as CVE-2017-8932: an arithmetic error in golang P256

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



using assembly from rust

Using assembly from rust

options:

1) "traditional"

(build time)

unfortunately:

- getting an assembler, c preprocessor, etc. is annoying on some platforms
- prevents inlining
- significant build-time cost and complexity
- function exit/entry ABI is platform-specific
  symbols defined in assembly need symbol mangling

Using assembly from rust

options:

~~1) "traditional"~~
2) transpile assembly to rust functions (containing inline assembly)

(development time)

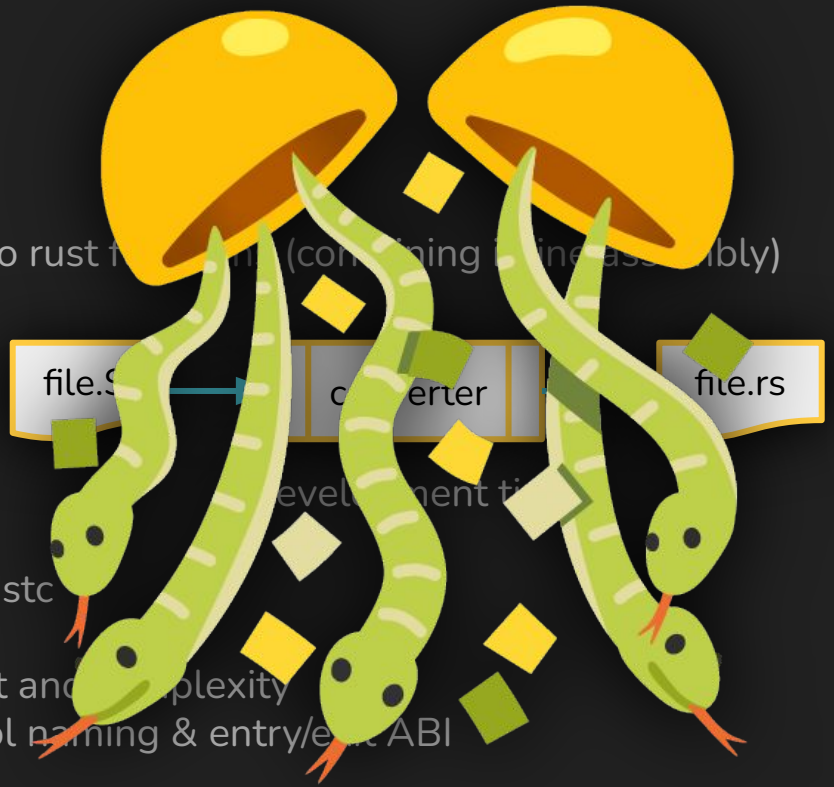
fortunately:

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

Using assembly from rust

options:

1) ~~"traditional"~~
2) transpile assembly to rust f… (con…ning i… …ine as… …bly)



…evel… …nent ti…

fortunately:

- build just requires rustc
- inlining works
- ~zero build time cost and … …plexity
- rustc handles symbol n…ming & entry/e… ABI

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

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

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

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

this leads to many macros in the crate: ~1500 in total

```
#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)),
```

```
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

goals:

- easy and fast to build
- for use with rustls - commonly-used cryptography for TLS
- competitive performance
- under common & permissive licenses

about graviola

achievements:

- easy and fast to build:
  - ascetic build requirements. just rustc. no build.rs, no proc-macros.
  - ~1 second build time
  - two dependencies: cfg-if & getrandom
- licensed under ISC + Apache2.0 + MIT-0
- 99.70% line coverage

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

Hashing

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

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 primitives

Hashing

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

AEADs

- AES-GCM
- chacha20-poly1305

new rust code, using intrinsics

limitations

- portability: ~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 v0.2.1
├── graviola v0.2.1
│   ├── cfg-if v1.0.0
│   └── getrandom v0.3.2
│       ├── cfg-if v1.0.0
│       └── libc v0.2.172
└── rustls v0.23.27
    └── …
```

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

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

signing - important for TLS servers

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

Signature verification

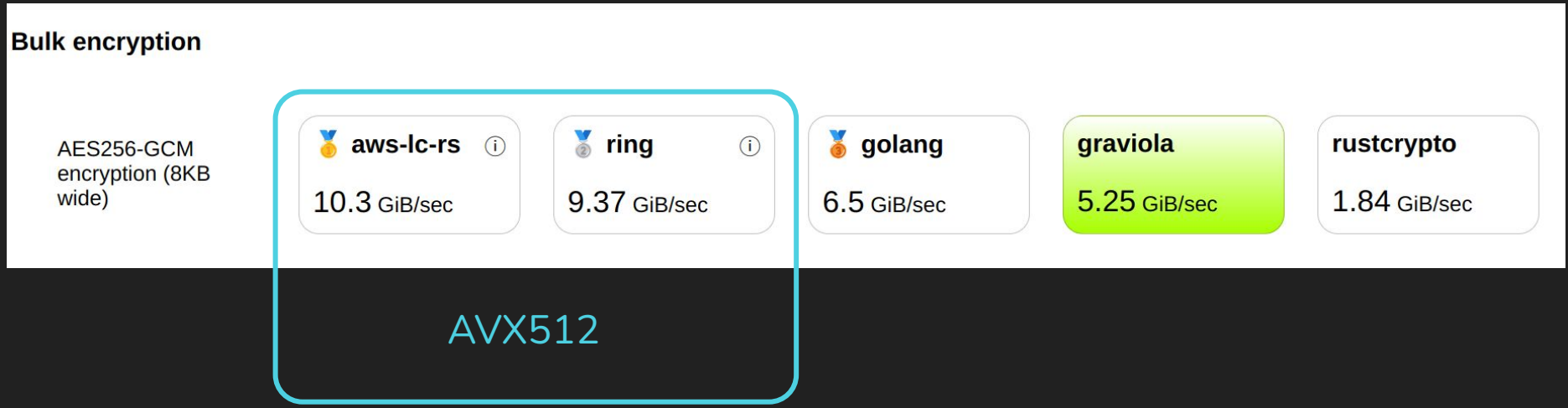
	🥇	🥈	🥉		
RSA2048 signature verification	**graviola** 85,182 sigs/sec	**aws-lc-rs** 78,069 sigs/sec	**ring** 66,076 sigs/sec	**golang** 46,786 sigs/sec	**rustcrypto** 7,385.7 sigs/sec
ECDSA-P256 signature verification	**aws-lc-rs** 27,338 sigs/sec	**graviola** 27,031 sigs/sec	**ring** 26,403 sigs/sec	**golang** 20,017 sigs/sec	**rustcrypto** 5,650.1 sigs/sec
ECDSA-P384 signature verification	**aws-lc-rs** 7,104.4 sigs/sec	**graviola** 5,145.4 sigs/sec	**golang** 2,316 sigs/sec	**ring** 1,840.3 sigs/sec	**rustcrypto** 1,244.9 sigs/sec

signature verification - important for TLS clients

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

key exchange - fundamental for all TLS, especially TLS1.3

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



AES-GCM - fundamental for all TLS

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

more on the page:

- aarch64 results (pretty similar story)
- versions tested
- hardware details
- full criterion reports

future work

~~RSA key generation~~

ML-KEM post-quantum key exchange

More architectures? (with reduced assurance!)

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

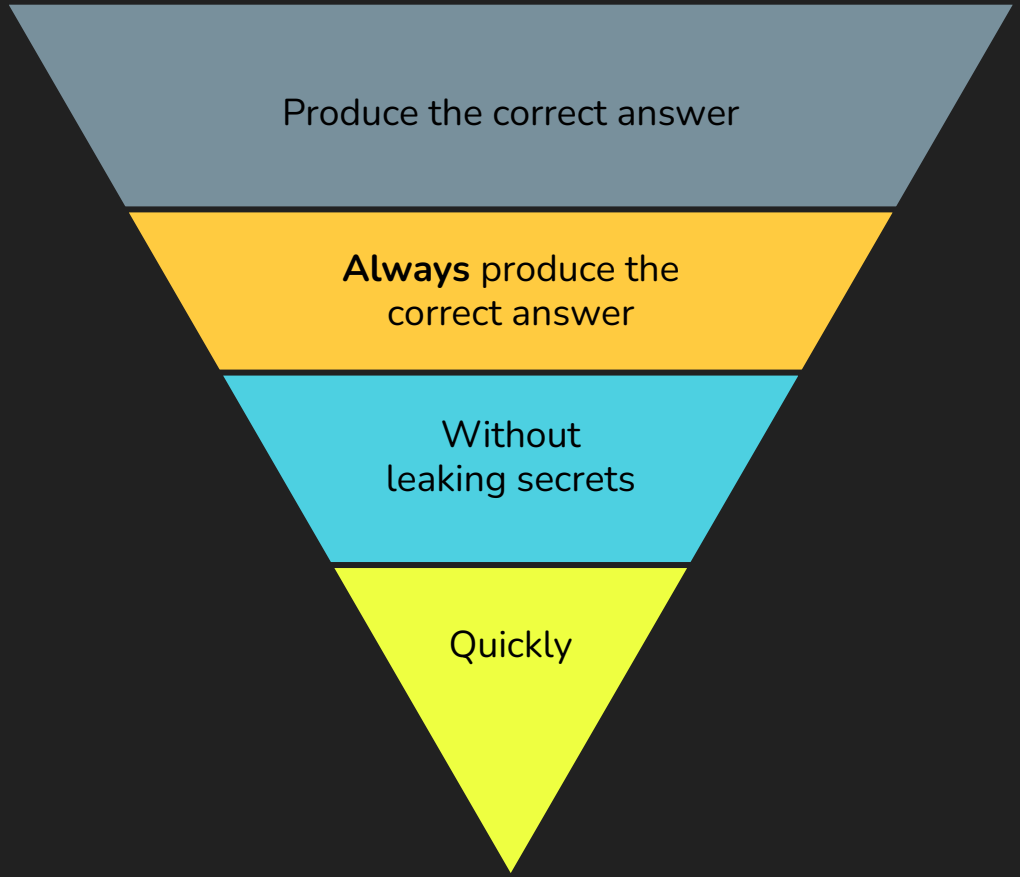
Slides: https://github.com/ctz/talks

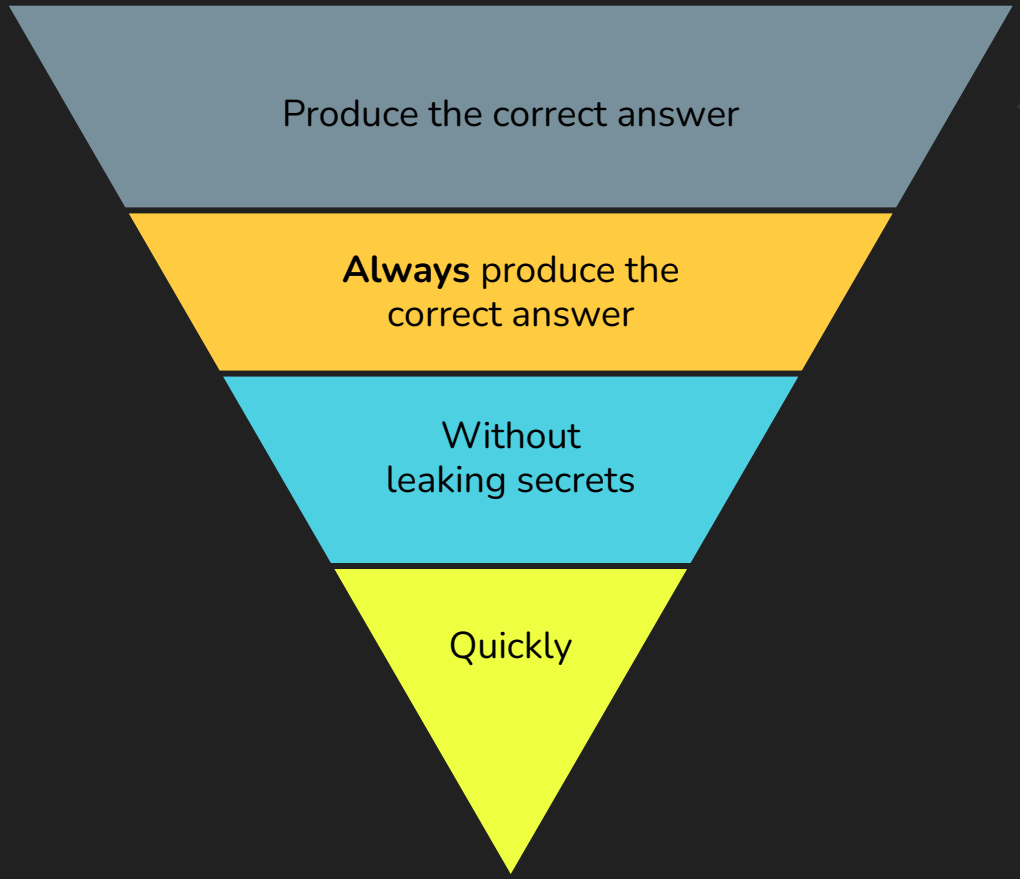
Illustrations: Google Emoji Kitchen

backup slides...

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

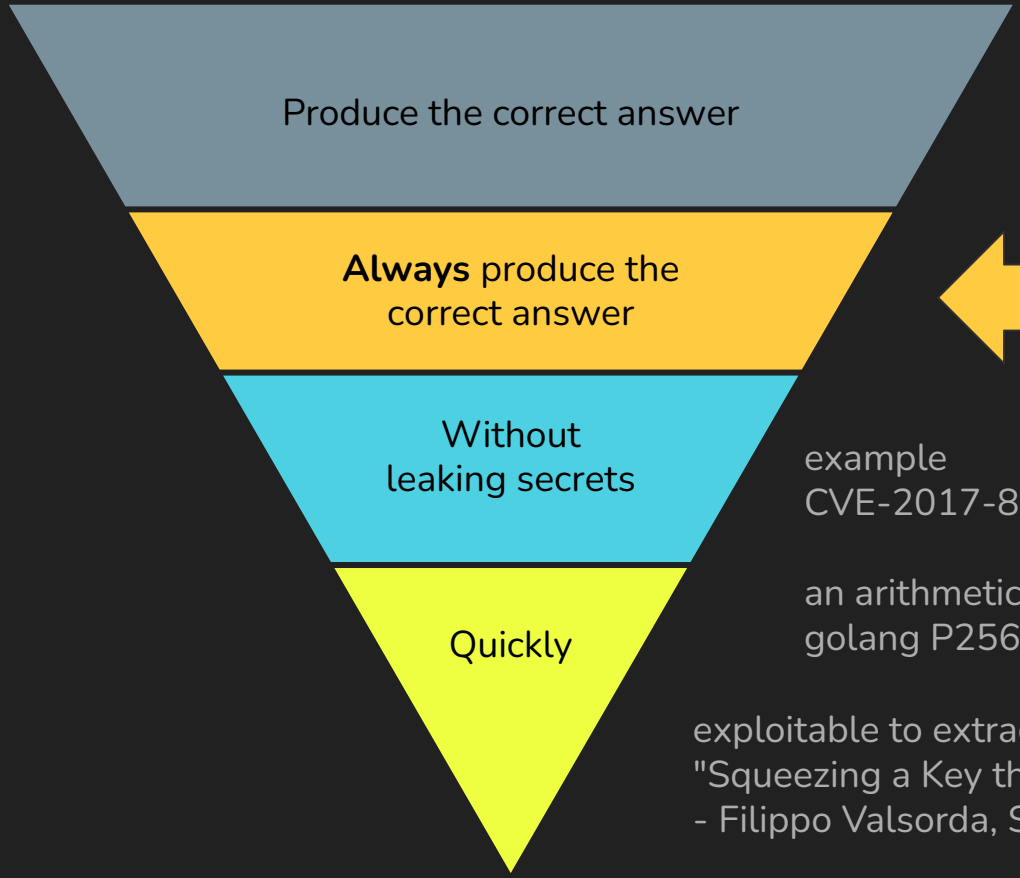
	aarch64 (ARM)	x86_64 (Intel)
RSA2048 signing	🥉 3rd	🥉 3rd
ECDSA-P256 signing	🥇 1st	🥇 1st
ECDSA-P384 signing	🥈 2nd	🥈 2nd
RSA2048 signature verification	🥉 3rd	🥇 1st
ECDSA-P256 signature verification	🥇 1st	🥈 2nd
ECDSA-P384 signature verification	🥈 2nd	🥈 2nd
X25519 key agreement	🥈 2nd	🥇 1st
P256 key agreement	🥈 2nd	🥉 3rd
P384 key agreement	🥈 2nd	🥈 2nd
AES256-GCM encryption (8KB wide)	💢 4th	💢 4th


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



basic testing

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



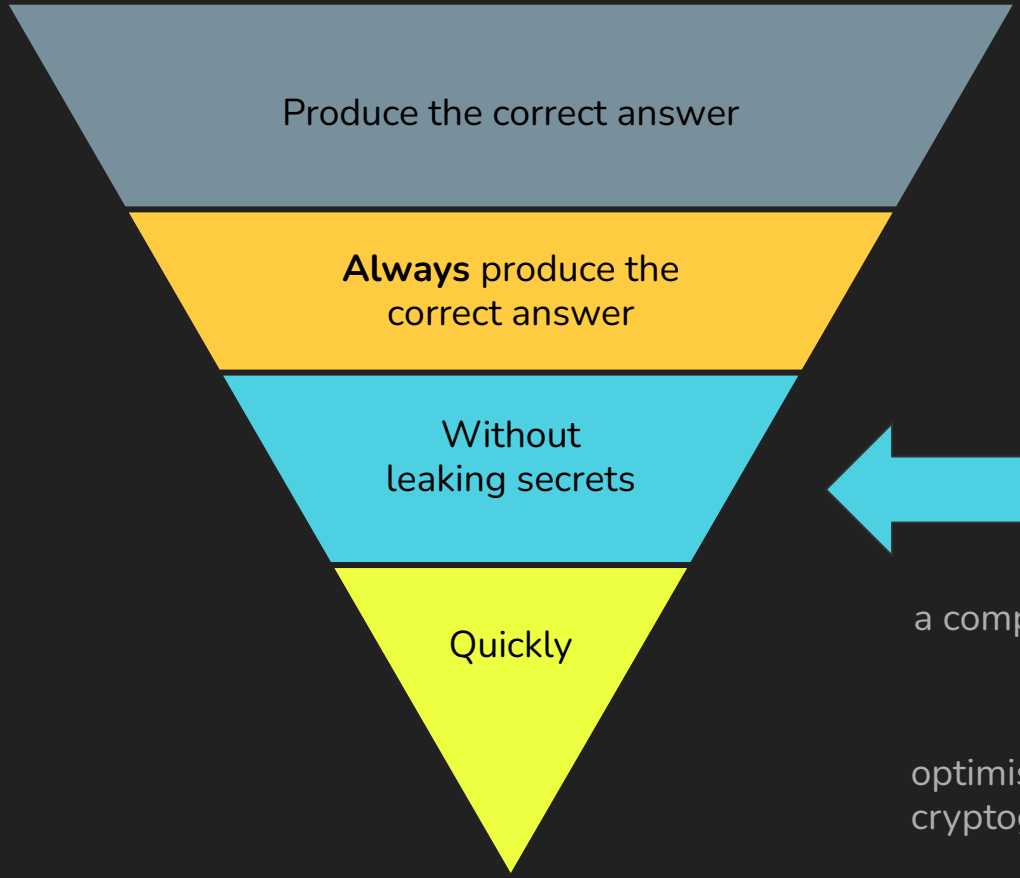
formal verification

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

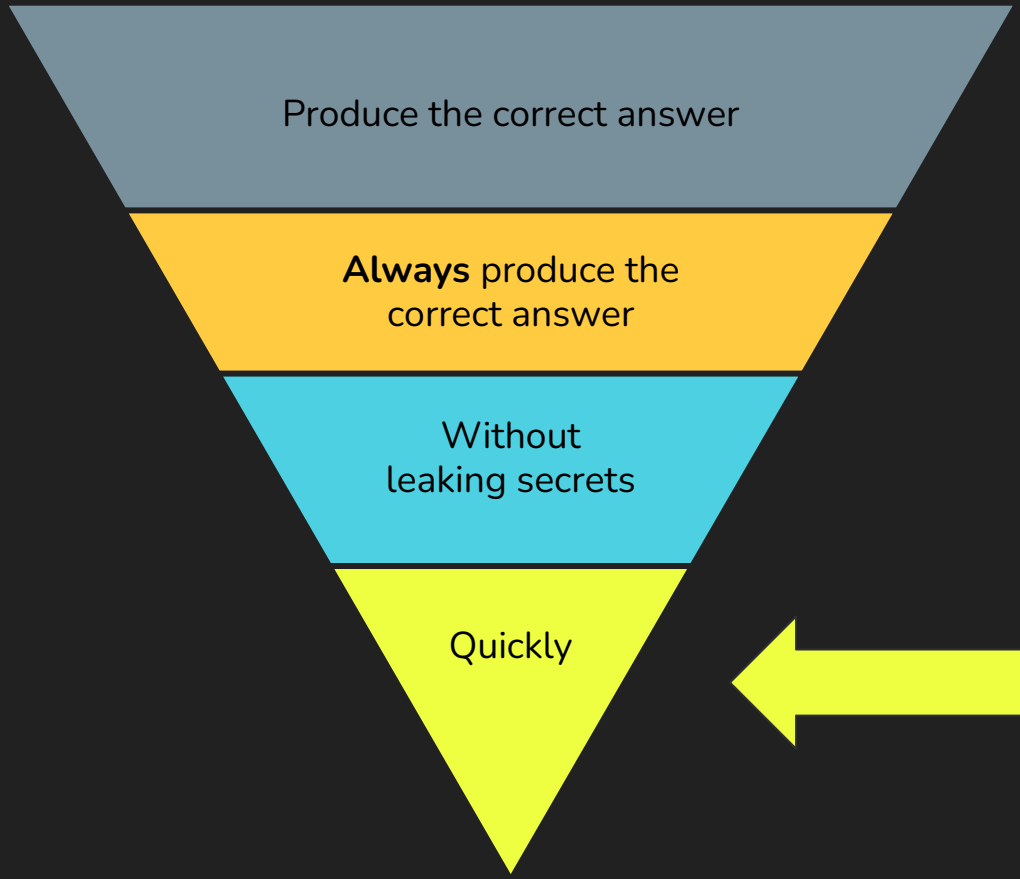
exploitable to extract private key data: see "Squeezing a Key through a Carry Bit" - Filippo Valsorda, Sean Devlin - Blackhat US 2018



side-channel safety

a complex and current problem

optimising compilers are bad for cryptography code



measure - understand - improve cycle

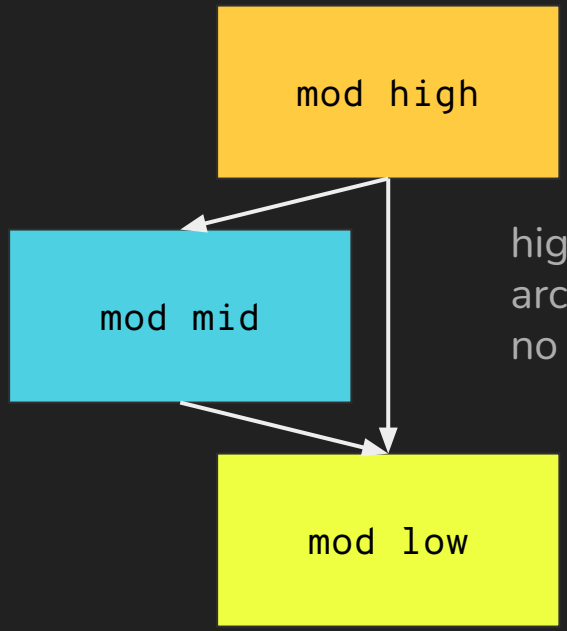
lots of literature on strategies to make things fast

lots of open-source, fast cryptography to learn from

matter of time, effort, and "mechanical sympathy"

performance analysis

graviola internal structure

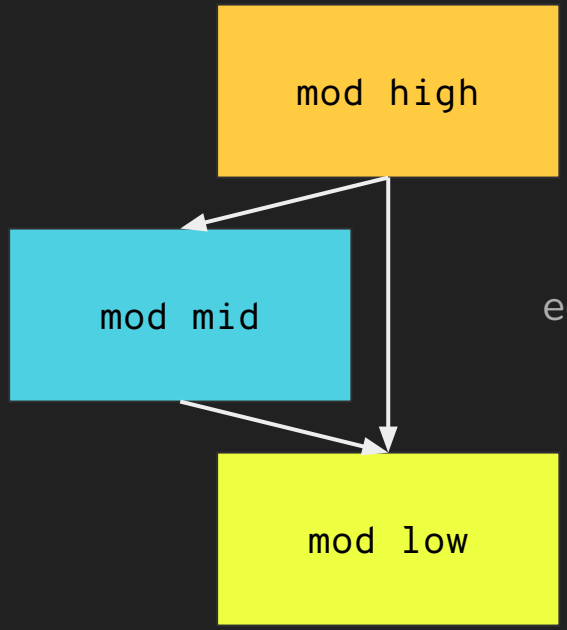


parsing and encoding
architecture agnostic
no unsafe

high-level arithmetic
architecture agnostic
no unsafe

low-level primitives
private
perhaps architecture-specific
unsafe allowed

graviola internal structure



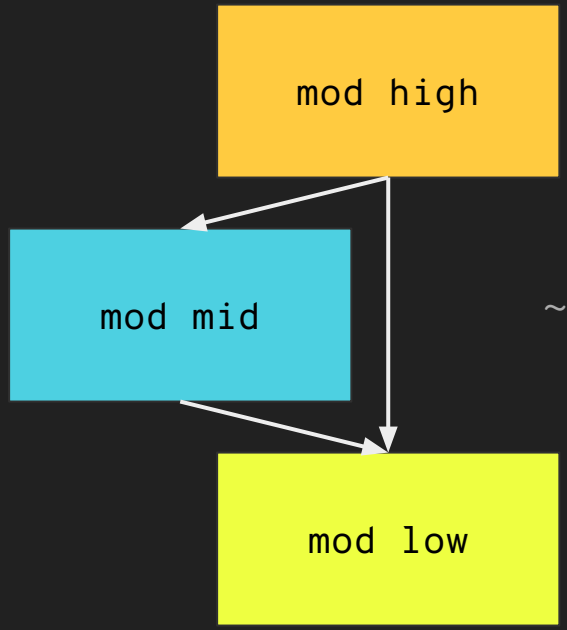
eg, ASN.1 decoding an RSA key

eg, RSA private operation

eg, multiplying large numbers

(mostly: safe functions backed by s2n-bignum)

graviola internal structure



~4400 lines

~4600 lines + ~19,000 lines of static data

~70,000 lines
of which
~63,500 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