SECURITY, PERFORMANCE: PICK ONE ?

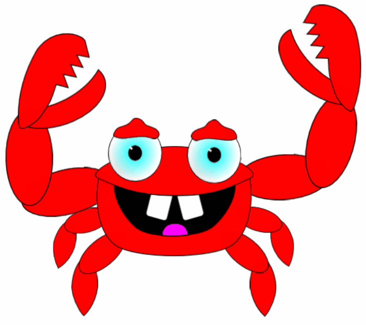
Pierre Chifflier
@pollux7

WHAT

- Defensive/Secure programming
- Needed for many programs
  - Web servers, IDS
- Often rejected for perf reasons
- We want performance **and** security
- And something we can maintain over time

WHO

- Debian Developer, Suricata contributor
- Head of the Detection Research lab at ANSSI
- Security, compilers and languages
- Write (rust) parsers for everything



DISCLAIMER

- This talk is my story, not a guide
  - Could be “story of a fight vs the compiler”
- Still, there are general rules/hints

CHOOSING A LANGUAGE

- Sometimes C is not the answer
  - You will *always* fail somewhere
  - Believe me, I’ve tried
- Parts in assembly
  - Could be fast, but a nightmare to maintain
- OCaml, Go, …
  - The perf-killer garbage collector

EXAMPLE: PARSING KERBEROS

```
KDC-REQ-BODY    ::= SEQUENCE {
        kdc-options             [0] KDCOptions,
        cname                   [1] PrincipalName OPTIONAL
                                    -- Used only in AS-REQ --,
        realm                   [2] Realm
                                    -- Server's realm
                                    -- Also client's in AS-REQ --,
        sname                   [3] PrincipalName OPTIONAL,
        from                    [4] KerberosTime OPTIONAL,
        till                    [5] KerberosTime,
        rtime                   [6] KerberosTime OPTIONAL,
        nonce                   [7] UInt32,
        etype                   [8] SEQUENCE OF Int32 -- EncryptionType
                                    -- in preference order --,
        addresses               [9] HostAddresses OPTIONAL,
        enc-authorization-data  [10] EncryptedData OPTIONAL
                                    -- AuthorizationData --,
        additional-tickets      [11] SEQUENCE OF Ticket OPTIONAL
                                       -- NOTE: not empty
```

ASN.1 DER ENCODING

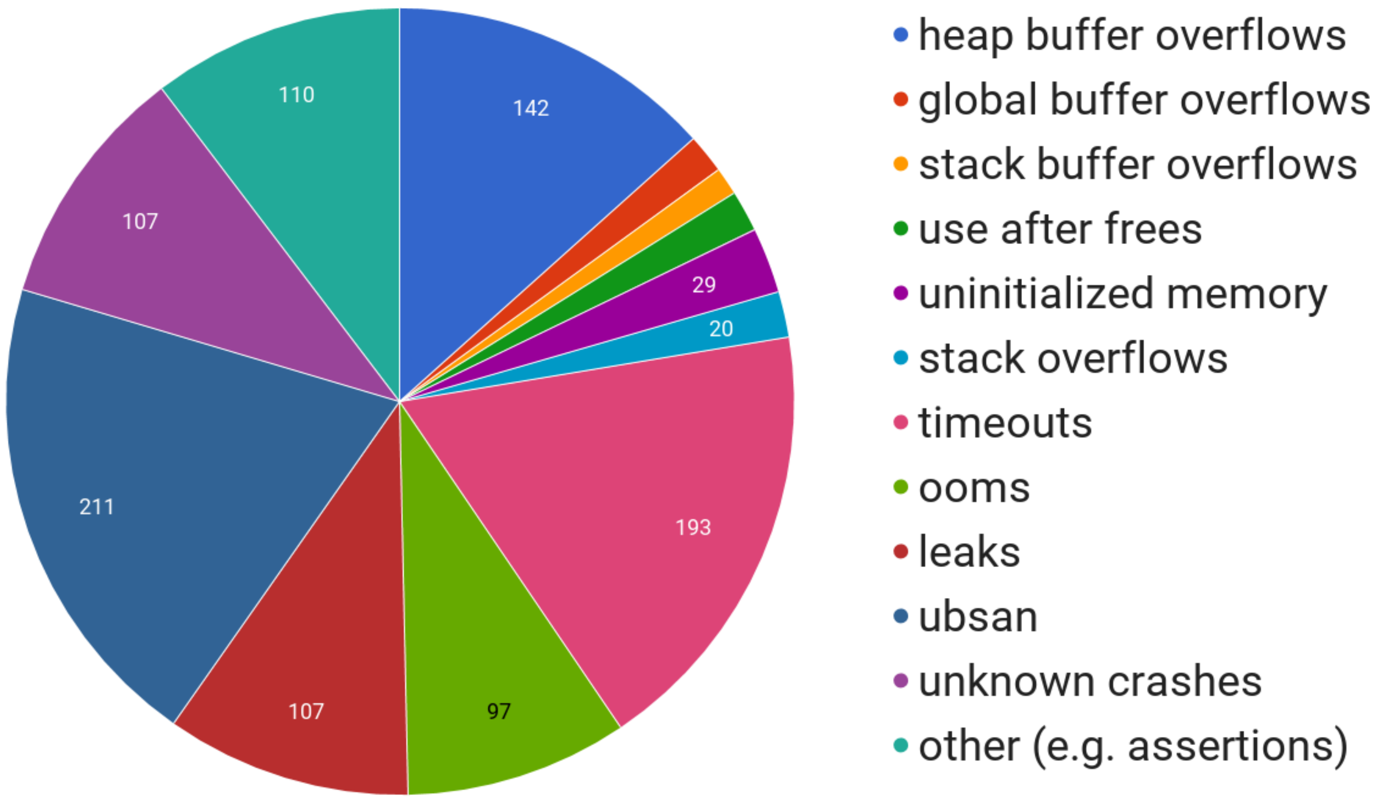
The 9 layers of DER Hell

- Lots of TLVs
- Highly recursive
- Infinite size integers
- Variable lengths

```
Ex: ASN.1 DigestInfo
30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20 XXXXXXXXXXXXXXXXXXXX

Tag             Length
30 ( SEQUENCE)  31
                   Tag                 Length
                   30 ( SEQUENCE)      0d
                                           Tag         Length
                                           06 ( OID)   09
                                                           OID
                                                           60 86 48 01 65 03 04 02 01
                                           Tag         Length
                                           05 ( NULL)  00
                   Tag                 Length
                   04 ( OCTET STRING)  20
                                           octet string ( SHA - 256 hash)
                                           XXXXXXXXXXXXXXXXXXXX
```

WRITING PARSERS IS HARD



(source: Google OSS-Fuzz)

RUST

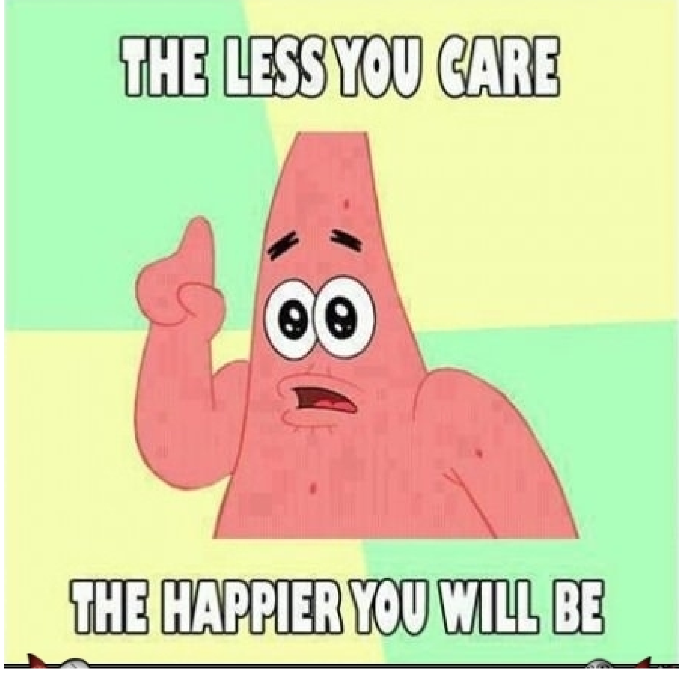
- Compatible with C
- Type safety
- Memory safety: non uninitialized values, etc.
- Thread safety: forces you to protect (lock) concurrent access
- Note: integer overflow/underflow still possible

RUST SPEED

- Fast, but not enough
- Can we do better
  - Keeping safety
  - Keeping some readability

MANDATORY WARNING

- First rule: don’t optimize
- Don’t bother one-time optimizations



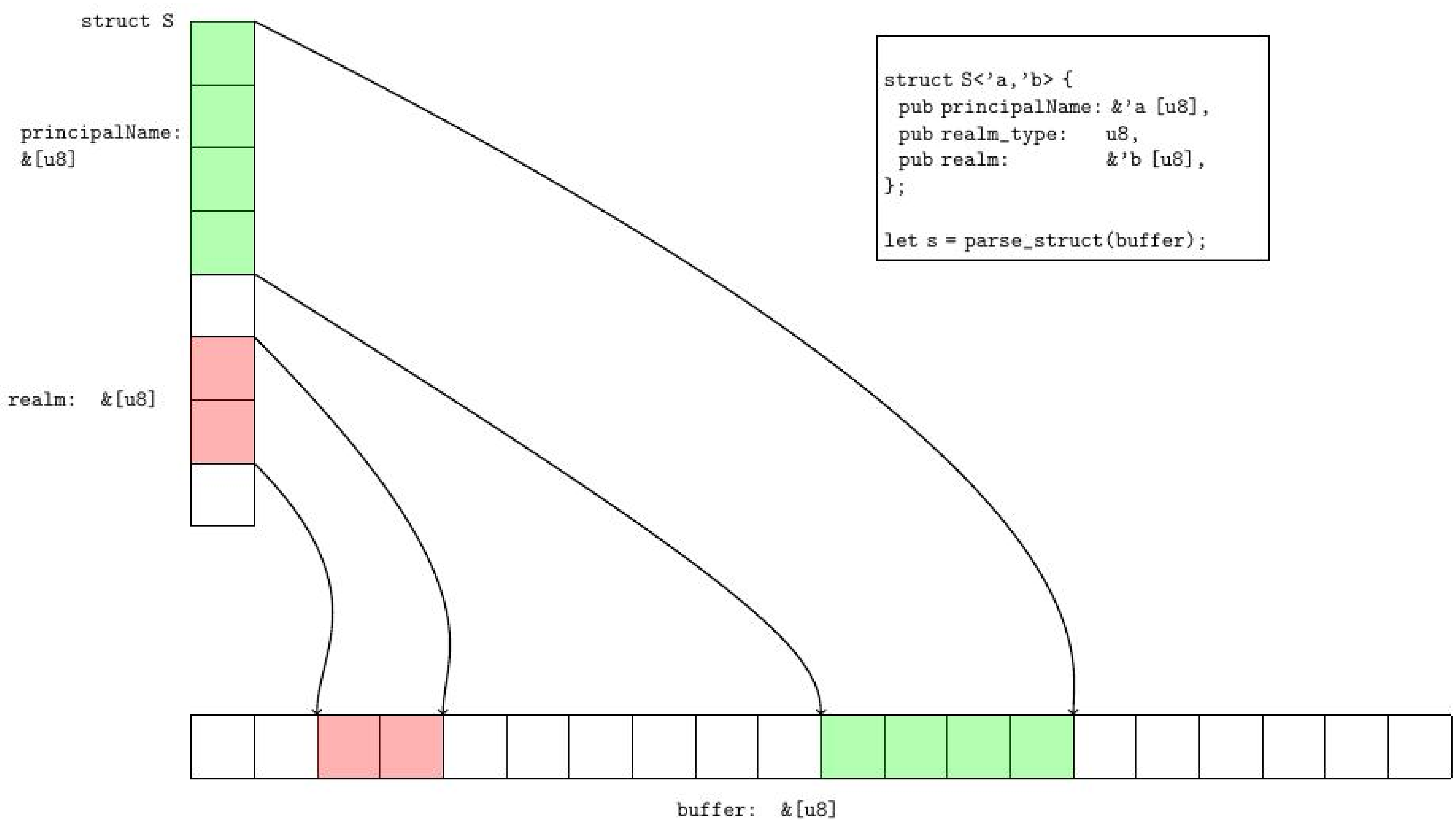
METHODOLOGY

- First: identifying slow points
- Use available tools
  - `cargo bench`
  - `perf`, `valgrind`
  - `flamegraph`
- One eye on the source code
- Also look at the produced binary code

ACTION 1: SOURCE CODE

- Algorithms first
- Zero-copy
  - Made possible thanks to *slices*
- Non-locking code
  - Borrow-checker and non-mutability help *a lot*

SLICES



struct S
principalName:
&[u8]
realm: &[u8]
buffer: &[u8]
struct S<'a,'b> {
pub principalName: &'a [u8],
pub realm_type: u8,
pub realm: &'b [u8],
};
let s = parse_struct(buffer);

RESULT 1

- messages of 305 bytes
- 1856 ns / message -> 156 MB/s (per thread)
- Fast, but we want more

ADDING INSTRUMENTATION: STEP 1

- Add to `Cargo.toml`:

```
[profile.release]
debug = true

[profile.bench]
debug = true
```

STEP 2: ADD BENCHMARKS

- Add a benchmark (`benches/b_krb5_parser.rs`):

```
static KRB5_TICKET: &'static [u8] = include_bytes!("../assets/krb5-ticket.bin");

#[bench]
fn bench_parse_ticket(b: &mut Bencher) {
    b.iter(|| {
        let res = parse_krb5_ticket(KRB5_TICKET);
        // use result !
        match res {
            Ok((rem,tkt)) => {
                assert!(rem.is_empty());
                assert_eq!(tkt.tkt_vno, 5);
            },
            _ => assert!(false),
        }
    });
}
```

STEP 3: COLLECT INSTRUMENTATION RESULTS

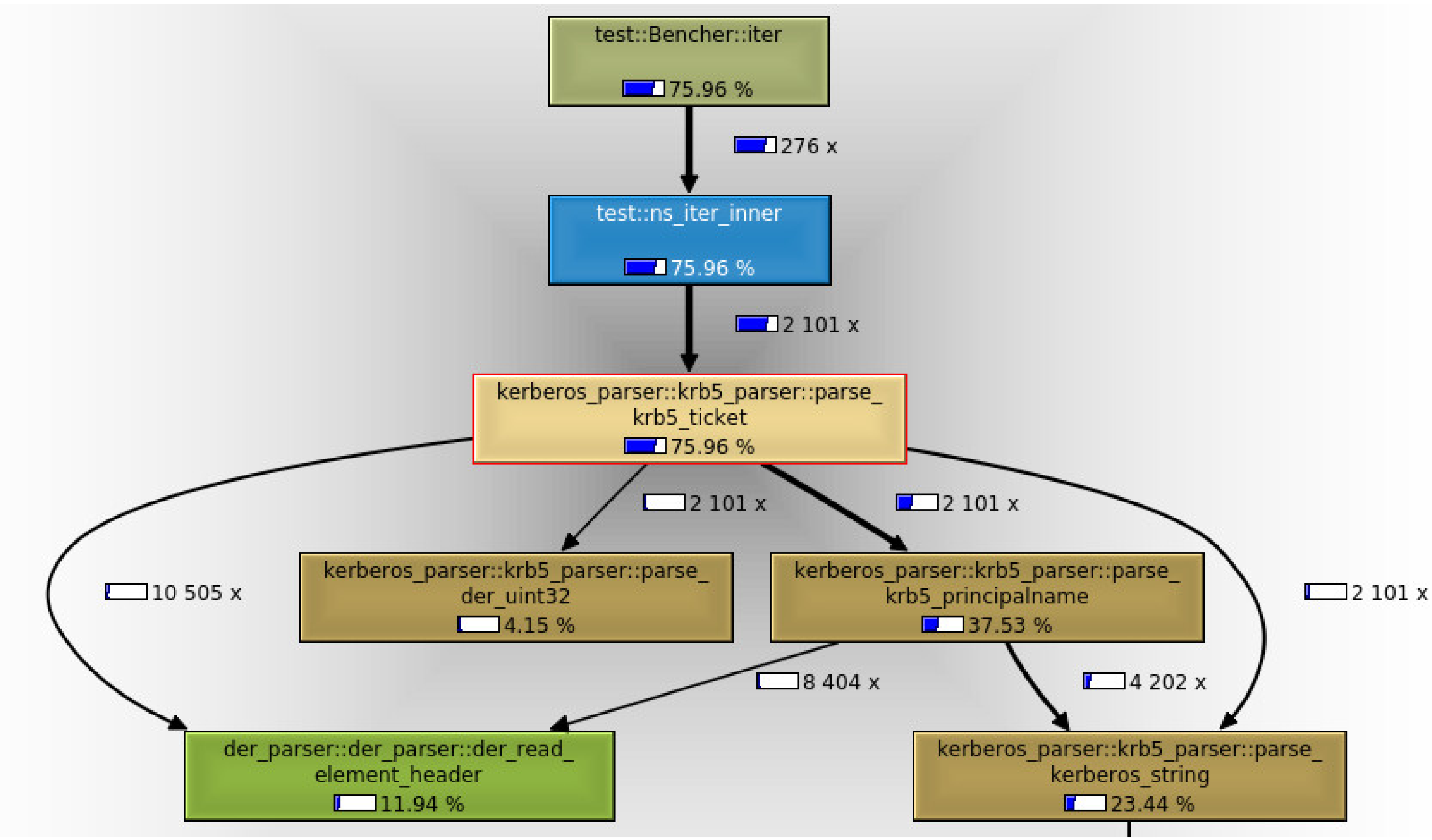
- Build the benchmark executable

```
cargo bench --no-run
```

- Use `valgrind` on it:

```
valgrind --tool=callgrind \
  --dump-instr=yes --collect-jumps=yes --simulate-cache=yes \
  ./target/release/b_krb5_parser-e84a853b88e37bef --bench bench_parse_ticket
```

PERFORMANCE GRAPHS



WHY IS IT SLOW ?

- Parts of the code are slow
  - too many tests
  - useless data copy
- Some structures do not fit in cache

ACTION 2: LOOK AT PRODUCED CODE

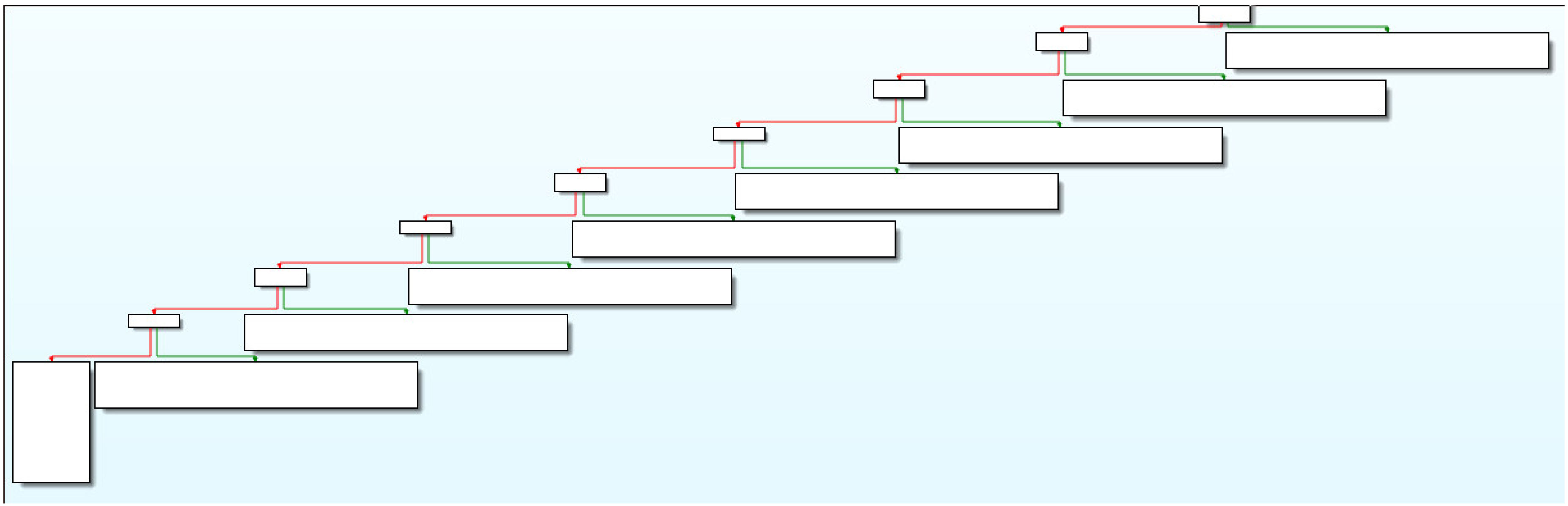
- Goal: try to identify and use efficient patterns
- Can bring huge speed improvements
- Time consuming
- Hard to find stable optimization patterns

COMPILER IS YOUR FRIEND .. OR NOT

EXAMPLE: READING AN U64

```
let u = (i[0] as u64) << 7 |
        (i[1] as u64) << 6 |
        (i[2] as u64) << 5 |
        (i[3] as u64) << 4 |
        (i[4] as u64) << 3 |
        (i[5] as u64) << 2 |
        (i[6] as u64) << 1 |
        (i[7] as u64);
```

EXAMPLE: READING AN U64



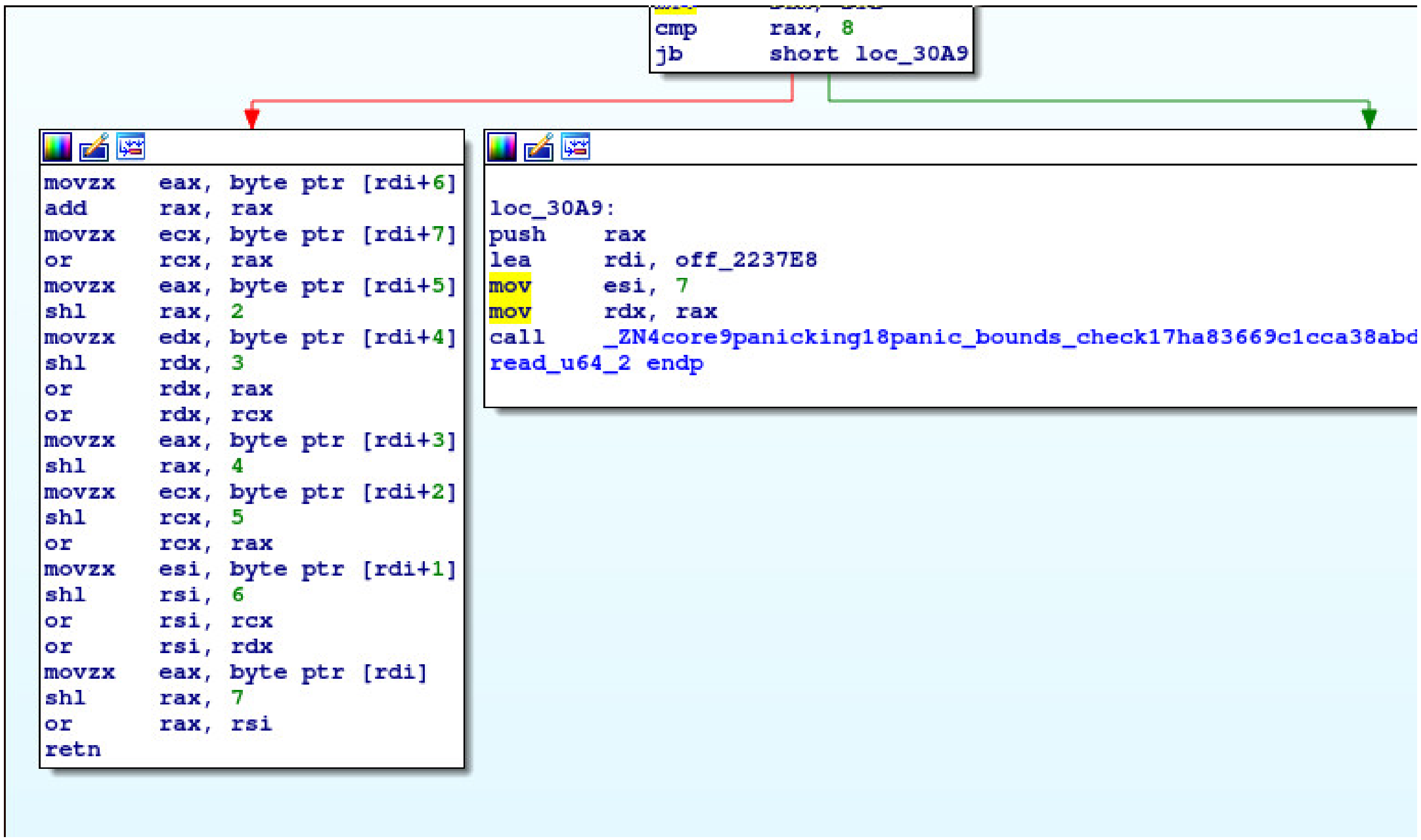
8 length tests

TEST #1: REORDERING

```
let u =
        (i[7] as u64)      |
        (i[6] as u64) << 1 |
        (i[5] as u64) << 2 |
        (i[4] as u64) << 3 |
        (i[3] as u64) << 4 |
        (i[2] as u64) << 5 |
        (i[1] as u64) << 6 |
        (i[0] as u64) << 7;
```

Read last bytes first

TEST #1: REORDERING



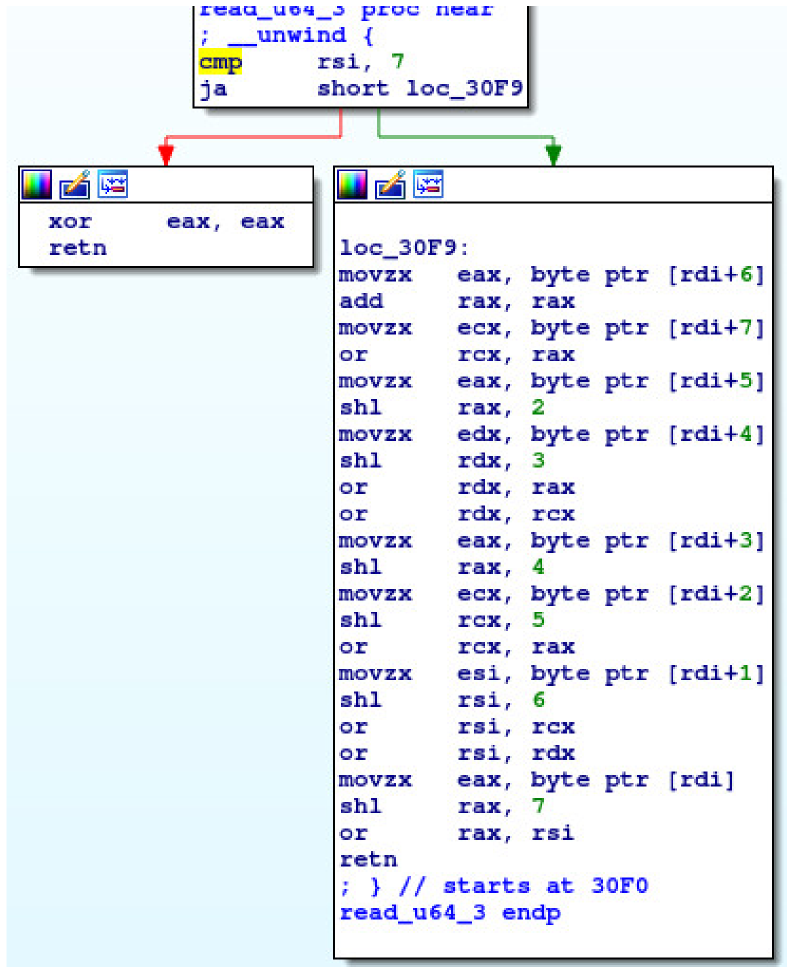
Better, but we still have a `panic` statement

TEST #2: ASSERTING/TESTING SIZE

```
if i.len() < 8 { return 0; }
let u = (i[0] as u64) << 7 |
        (i[1] as u64) << 6 |
        (i[2] as u64) << 5 |
        (i[3] as u64) << 4 |
        (i[4] as u64) << 3 |
        (i[5] as u64) << 2 |
        (i[6] as u64) << 1 |
        (i[7] as u64);
```

- Compiler uses the info from the test
- No need to reorder

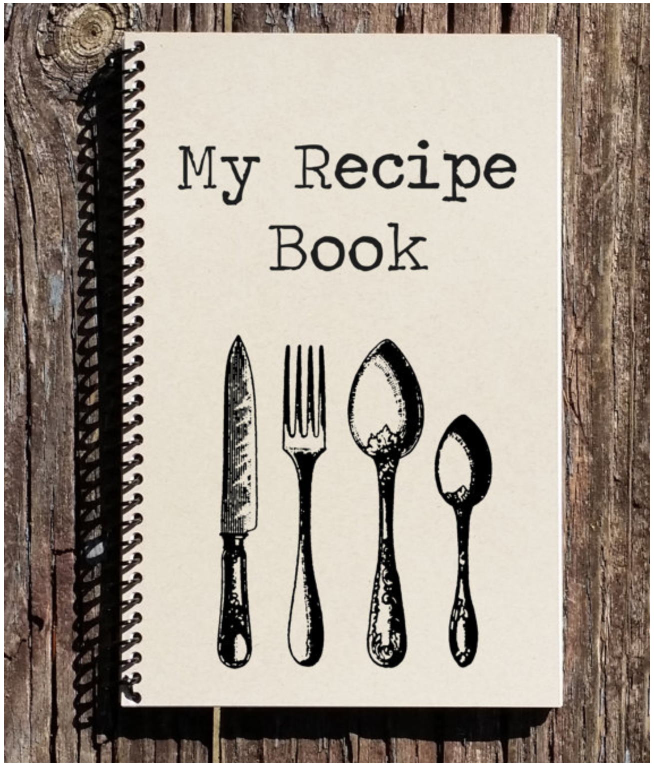
TEST #2: ASSERTING/TESTING SIZE



LESSONS FROM THAT EXAMPLE

- Compiler is smart
- But not enough to infer all information
- Sometimes *adding* code makes result faster
  - Some tests/asserts have to be *explicit*
- We can get efficient code without using `unsafe` or assembly

SOME OTHER TIPS



PACKED ENUMS

Representing a packed enum:

```
#[repr(u8)]
pub enum Foo {
    Value1 = 1,
    Value2 = 2,
    ...
```

However:

- `match` is slow
- conversions to/from `u8` are implemented as either
  - function calls (slow)
  - memory casts (unsafe)

THE NEWTYPE PATTERN

```
pub struct Foo(pub u8);
```

- Type-safe, cost-free abstraction
- Free conversions
  - except if you forget the `pub` keyword!
- Compile time increases
- Values have to be declared as associated constants

```
impl Foo {
    pub const Value1 : Foo = Foo(1);
```

ALLOCATIONS

- Allocations are slow
- Prefer the stack
  - Avoid `Box` and `Vec`
  - You can use variable-length data-types on stack
  - Drawback: calls to `memcpy`

STRUCTURES

- Keep as much as possible in cache
  - Use small structs
  - Make sure they fit in cache
- Check using `valgrind`

CODE

- Keep as much as possible in cache
- Keep as much as possible in registers
- Use reentrant, pure functions (no side-effects)
- Avoid locks and global structures
  - locks are slow!

CODE

- Write *linear* code
  - Avoid instructions cache misses
- *nom* helps a lot (macros)
  - Possible problem: cyclomatic complexity

```
warning: the function has a cyclomatic complexity of 231
   --> src/krb5_parser.rs:303:1
```

AUTOMATIC VECTORIZATION

```
let len = min(min(a.len(), b.len()), c.len());
for i in 0..len {
    c[i] = a[i] + b[i];
}
```

Code is not vectorized:

```
cmp     rsi, r10
jae     .LBB0_7
cmp     rsi, rcx
jae     .LBB0_8
cmp     rsi, r9
jae     .LBB0_9
mov     eax, dword ptr [rdi + 4*rsi]
add     eax, dword ptr [rdx + 4*rsi]
mov     dword ptr [r8 + 4*rsi], eax
lea     rax, [rsi + 1]
mov     rsi, rax
...
```

AUTOMATIC VECTORIZATION

```
let len = min(min(a.len(), b.len()), c.len());
let (a,b,c) = (&a[..len], &b[..len], &mut c[..len]);
for i in 0..len {
    c[i] = a[i] + b[i];
}
```

Code is vectorized:

```
movdqu xmm0, xmmword ptr [r10 + 4*rsi]
movdqu xmm1, xmmword ptr [r10 + 4*rsi + 16]
movdqu xmm2, xmmword ptr [rdx + 4*rsi]
paddd xmm2, xmm0
movdqu xmm0, xmmword ptr [rdx + 4*rsi + 16]
paddd xmm0, xmm1
...
```

However:

- Make sure instructions apply to **blocks** of data

MISC PATTERNS

- Avoid `Box<Trait>`, prefer `&mut Trait`
  - former is 2 pointers (and extra checks)
- Use iterators
  - they can spare some more bounds checks

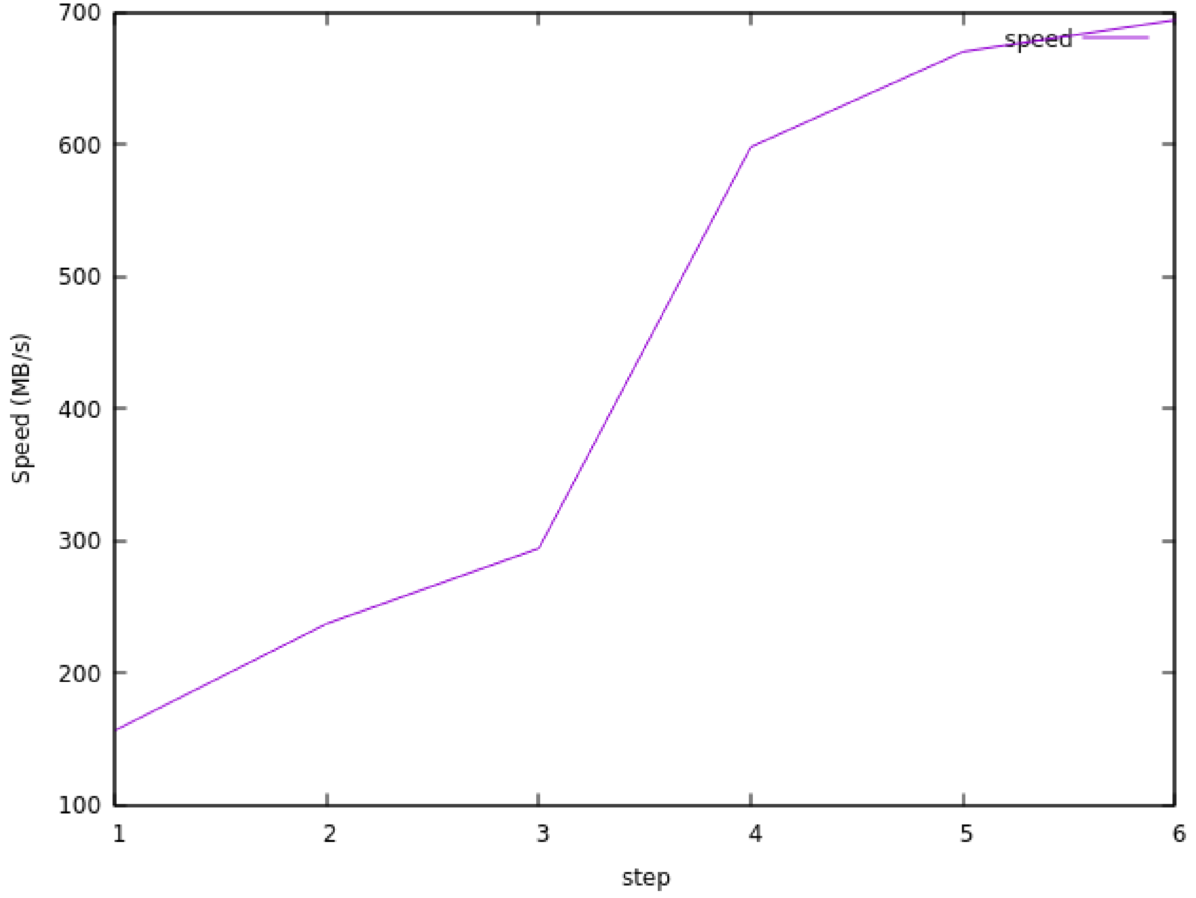
LINKER

- Use LTO (Link-time optimization)
  - Bigger executable, generally faster
  - Better than using `#[inline(always)]`
- Test PGO (Profile-guided optimization)
  - Variable results
  - Sometimes really good

CHECKING WE STILL HAVE SECURITY

- Compiler is not (always) your friend
  - Undefined behaviors
    - Integer underflows/overflows
    - Removed in release mode, but can be added
  - Removed tests
  - Removed calls (*e.g.* `memset`)
  - `panic / assert` inserted or remaining
- Use Compiler Explorer (https://rust.godbolt.org/)
- Use `cargo fuzz`

RESULTS (KERBEROS)



- Comparison: my C implem. is ~ 500 MB/s

CODE

- Kerberos, SNMP, IKEv2, TLS, Radius, etc.
- Rusticata project (and all parsers): https://github.com/rusticata
- Rust code push on crates.io
- Most parsers merged in Suricata git repo (4.1 ?)

LESSONS

- Guiding the compiler is efficient
- And easier to maintain than writing assembly
- Always check that the tests are present after optimizing
- Are the optimizations stable ?
  - Not guaranteed, but in practise yes
  - Sometimes look like voodoo