Understanding Code: The Scientific Method for Approaching Code

Gabriel Parmer
gparmer@gwu.edu
The George Washington University (GWU)

Systems and Security Lunch (SSL) Presentation

Systems Research Timeline

- Idea! Requires implementation/evaluation
- Identify system with design amenable to your goals
- Unpack (*amazing looking*) system
- Build and test new system
- **Read through system's code**
- Modify (distinctly less amazing) existing code
- Debug, `goto pain`...iterate
- (9 months and much blood later)
  Evaluate *shiny* new (disgusting-looking) system
- Publish world-changing research

Goal: Productivity in a Large Code-base

Today: how to effectively read through system's code

- **scientific method** for reading code
- the onion model for code traversal

Analogy...

Speed Dating: The typical method

Getting to know someone (Surface questions)

- Hey, how you do'in
- How 'bout that weather?
- . . . and those red sox/nationals/sport team?

A reliable way to find a compatible partner?

Speed Dating: The tactless method

Just ask better (uncomfortable) questions? (Deep questions)

- How do you vote?
- What's your credit score?
- What are your dreams?
- What are your values?
- `diff values actions | wc -l`
- `diff dreams direction | wc -l`

Done! 6 questions to determine the rest of your life!

A reliable way to find a compatible partner?

Give up! Succumb to biology

Forget dating, go to the *dive bar*!

#1 and only criteria: *Superficial Attractiveness*

A reliable way to find a compatible partner?

Dating Works (?)

$$\begin{aligned} \text{Surface questions} &\neq \text{good compatibility test} \\ \text{Deep questions} &\neq \text{good compatibility test} \\ \text{Superficial attractiveness} &\neq \text{good compatibility test} \\ ??? &= \text{good compatibility test} \end{aligned}$$

I hear people get married...

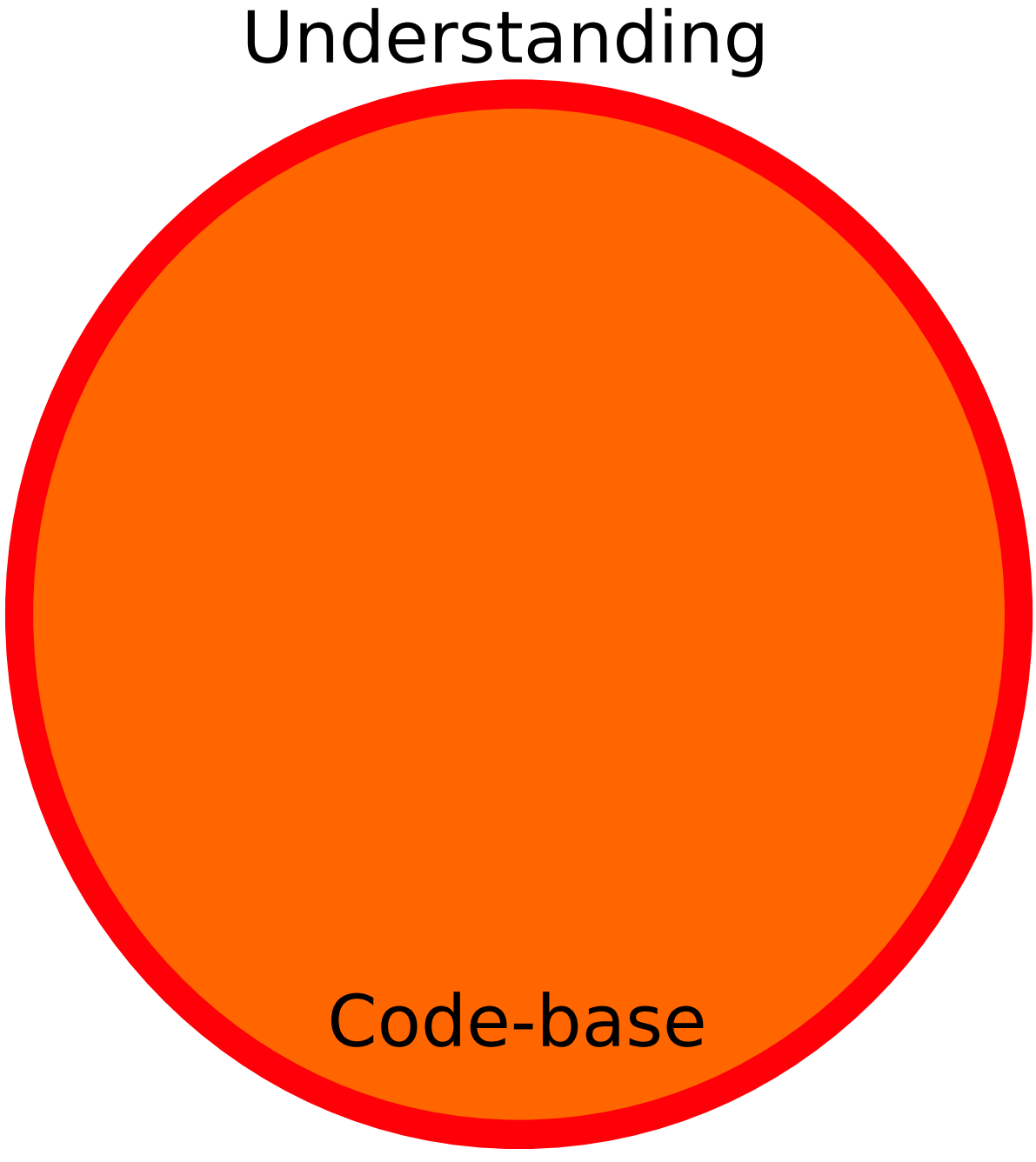
Analogy

Getting to know a partner	=	Understanding a system
Surface questions	=	"reading the code"
Deep questions	=	characterizing behavior (experiments)
Superficial attractiveness	=	design documents (diagrams, papers, concepts) . . . window-dressing and lies

These things *are important*

. . . but not great for *understanding* a system

Understanding Code: A diagram



The Speed Dating of Reading Code

Some superficial attractiveness:

- high-level structure: *design* documents
- paper-level understanding

Some small talk:

- directory/build structure
- code structure
  - more variant in C/Python/..., less-so in Java/Haskell
- code rules/conventions (style guides)
- Most important: naming conventions

Aside: efficiency is important – learn your tools

- lxr (web-based cross references)
- TAGS, GTAGS
- grep
- emacs (iswitch, M-/, M-. w/ TAGS, goto-line, C-s+C-w)

Getting to know you: Example

CBUFs= COMPOSITE predictable, efficient data passing

- data shared $\text{comp}_0 \longleftrightarrow \text{comp}_1$ in 300 cycles

Examples

- `doc/design/*` = high-level design
- `src/components/include/cbuf.h` = common-case functions
- inlined functions = fast-path
- `src/components/interface/cbuf_c/*.c` = non-fast-path, client code
- `src/components/interface/cbuf_c/cbuf_c.h` = manager interface
- `src/components/implementation/cbuf_c/naive/*` = manager/server code
- `cbuf_*` = client-facing interface functions

Reading code: Naive approach

What is your algorithm for reading code?

Reading code: Naive approach

Depth-First Search (DFS)

1. inspect function
2. see function calls?
    - no called functions? return to previous fn
3. see called function, goto 1

Reading code: Naive approach

Depth-First Search (DFS)

1. inspect function
2. see function calls?
    - no called functions? return to previous fn
3. see called function, goto 1

This is small talk

- mechanical, non-engaging

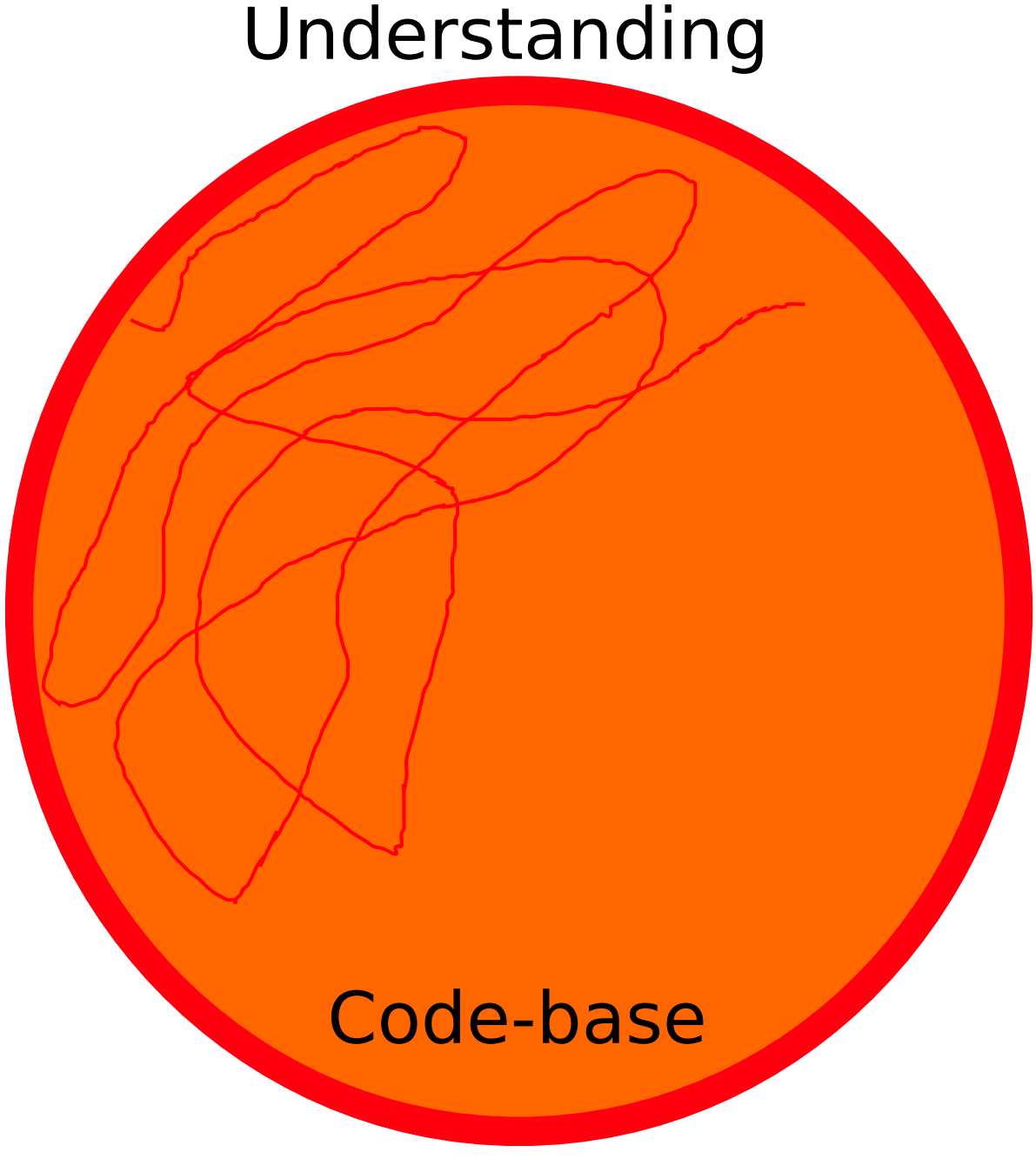
Reading code: Naive approach

Depth-First Search (DFS)

1. inspect function
2. see function calls?
   - no called functions? return to previous fn
3. see called function, goto 1

Why doesn't this work? Better way?
How about just use BFS?

DFS $\neq$ Understanding



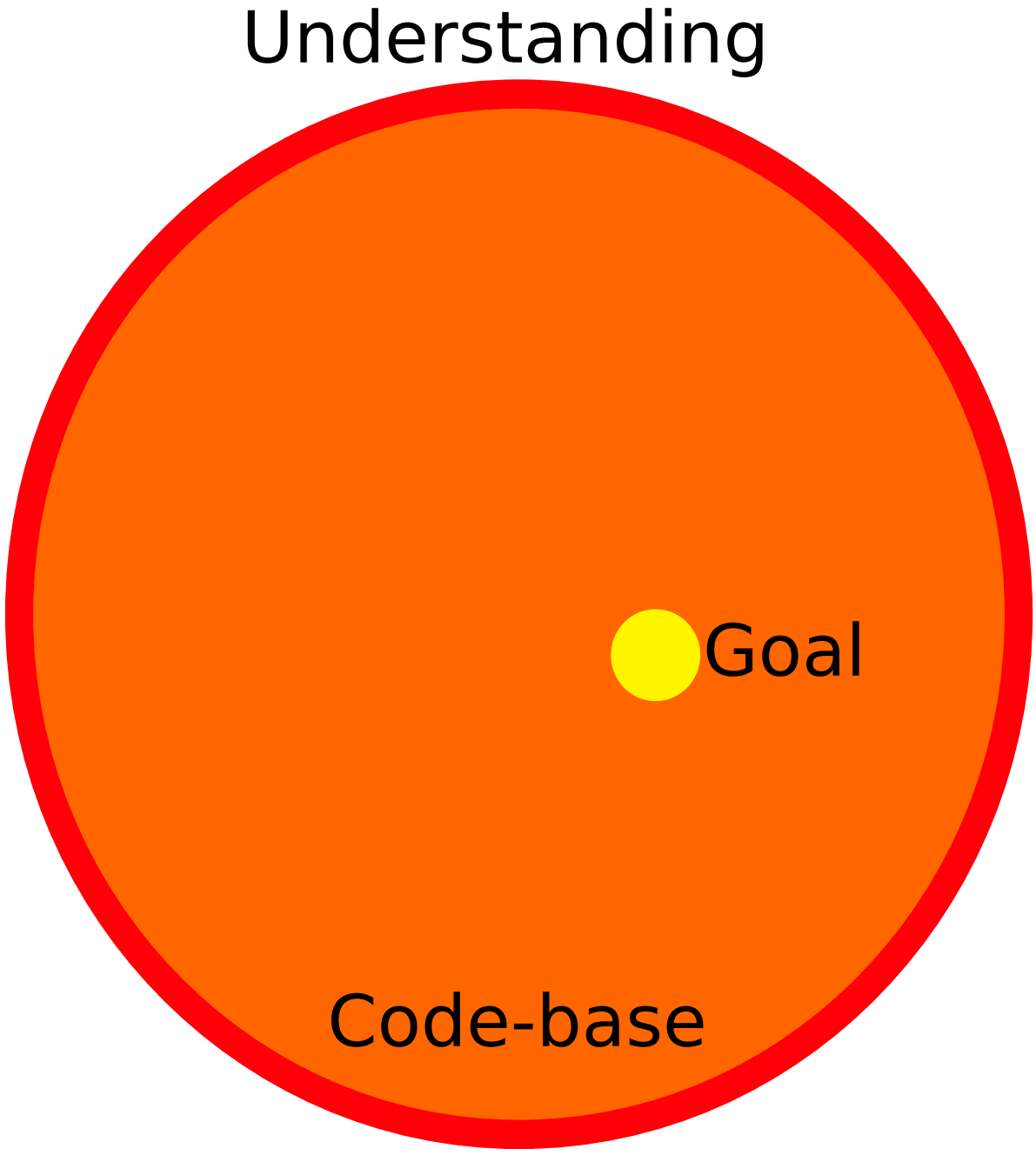
Scientific Method for Reading Code

1. **Goal**: What are you trying to learn?
   - "how CBUFs efficiently shared between components?"
   - no goal = waste of your time

2. **Hypothesis**:
   A strong statement about the current code of interest
   - "CBUFs are optimized for a cached common-case to avoid invocations"
   - "the cached case is handled only in `cbuf.h`"
   - "the non-cached case handed off to `*_slow` functions"
   - "`*_slow` functions call `cbuf_mgr` → maps CBUFs"

3. Confirm/refute hypothesis
   - confirm = next hypothesis → goal
   - refute = back up, reformulate hypothesis

Understanding Code: Goal-directed



Hypothesis: Guided, Deepening Understanding

Ideal hypothesis stated

- relative to goal
- relative to high-level design
- informed by *behavior/tests*

hypothesis = f(design, goal, behavior, understanding of code)

Hypothesis: Guided, Deepening Understanding

Ideal hypothesis stated

- relative to goal
- relative to high-level design
- informed by *behavior/tests*

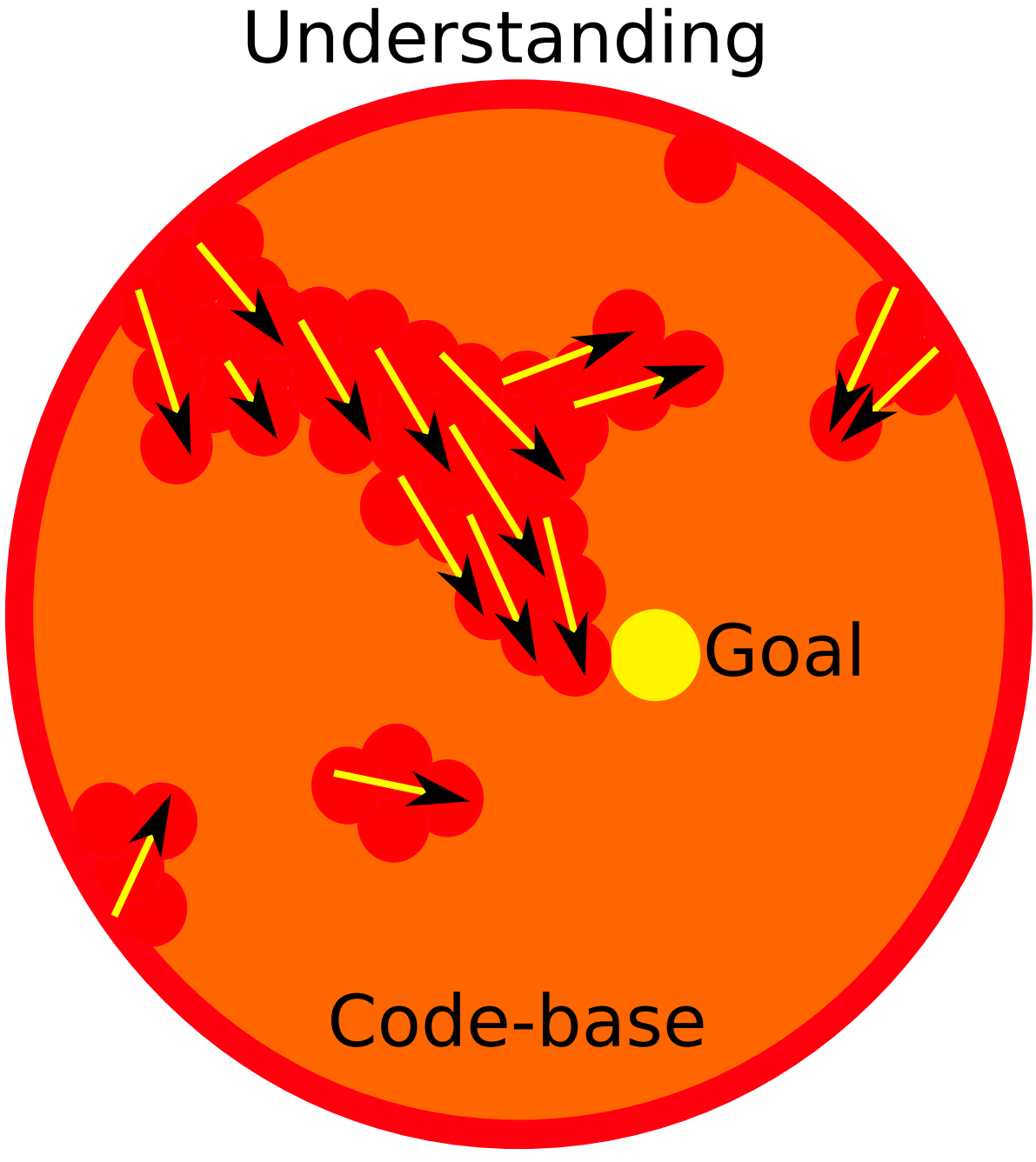
hypothesis $= f$(design, goal, behavior, understanding of code)

Hypothesis is both

- a sub-goal in understanding – direct code-observation
- a mapping: high-level concepts $\leftrightarrow$ code in front of you

$\text{hypothesis}_0 = f(\text{goal, behavior, design, surface questions})$
$\text{hypothesis}_n = f(\text{goal, behavior, design, hypothesis}_{n-1})$

Understanding Code: Hypothesis-guided



Hypothesis: Your Own Little Spanish Inquisition

Hypothesis are *abstraction for understanding code*

- write them down
- these are your current understanding of the code
- they abstract details of code into high-level statements
  - ...that humans can remember/understand
  - you *will* forget the code
  - you *should* forget the code

$\rightarrow$ *abstraction for reading code*

How do we confirm hypothesis?

Hypothesis: Your Own Little Spanish Inquisition

Hypothesis are *abstraction for understanding code*

- write them down
- these are your current understanding of the code
- they abstract details of code into high-level statements
  - ...that humans can remember/understand
  - you *will* forget the code
  - you *should* forget the code

$\rightarrow$ *abstraction for reading code*

How do we confirm hypothesis?

- we don't
- we gain *confidence* in them
- they could go wrong at any point

In the Trenches, Looking at Code

I'm reading code, I have hypothesis and a goal...
...how do I direct my search through the code?

DFS? BFS?

In the Trenches, Looking at Code

I'm reading code, I have hypothesis and a goal...
...how do I direct my search through the code?

Onion model

- layers of abstraction
- layers of complexity
- important to distinguish between layers

Example

```
static inline void *
cbuf2buf(cbuf_t cb, int len)
{
        u32_t id;
        struct cbuf_meta *cm;
        union cbufm_info ci;
        void *ret = NULL;
        long cbidx;
        if (unlikely(!len)) return NULL;
        cbuf_unpack(cb, &id);

        CBUF_TAKE();
        cbidx = cbid_to_meta_idx(id);
again:
        do {
                cm = cbuf_vect_lookup_addr(cbidx);
                if (unlikely(!cm || cm->nfo.v == 0)) {
                        if (__cbuf_2buf_miss(id, len)) goto done;
                        goto again;
                }
        } while (unlikely(!cm->nfo.v));
        ci.v = cm->nfo.v;

        if (unlikely(!(cm->nfo.c.flags & CBUFM_TMEM))) goto done;
        if (unlikely(len > PAGE_SIZE)) goto done;
        ret = ((void*)(cm->nfo.c.ptr << PAGE_ORDER));
done:
        CBUF_RELEASE();
        assert(lock_contested(&cbuf_lock) != cos_get_thd_id());
        return ret;
}
```

Onion model: Everyone's Crying

Implication: When reading code, question

- is invoked function at same level
  ...or peeling off another layer?
- understand current level before next
  - get an intuition for current layer
  - confirm hypothesis about current layer
  - understand how next level is used
  - form hypothesis about next layer(s)
    - "that function must do x"
  - ROT: no hypothesis about fn? Don't go into it!
  - ROT: stay in the same file/source directory

The onion analogy – lots of weeping when peeling abstractions

- new peel, more crying $\rightarrow$ peel less!
- we're learning to peel underwater $\rightarrow$ peel better!

Brains have a Small Register File

Even within a peel, we have a problem:
...we suck at reading code

- Human short-term memory: $7 +/- 2$ items
  - each conditional (-1), loop (-1), variable (-1)
- CS = abstraction; use this!

Brains have a Small Register File

Even within a peel, we have a problem:
...we suck at reading code

- Human short-term memory: 7 +/− 2 items
  - each conditional (-1), loop (-1), variable (-1)
- CS = abstraction; use this!

An onion peel:

- code contains more than 7 "things"
- at a given abstraction level *what can you instantly forget?*

Example: what to ignore?

```
static inline void *
cbuf2buf(cbuf_t cb, int len)
{
        u32_t id;
        struct cbuf_meta *cm;
        union cbufm_info ci;
        void *ret = NULL;
        long cbidx;
        if (unlikely(!len)) return NULL;
        cbuf_unpack(cb, &id);

        CBUF_TAKE();
        cbidx = cbid_to_meta_idx(id);
again:
        do {
                cm = cbuf_vect_lookup_addr(cbidx);
                if (unlikely(!cm || cm->nfo.v == 0)) {
                        if (__cbuf_2buf_miss(id, len)) goto done;
                        goto again;
                }
        } while (unlikely(!cm->nfo.v));
        ci.v = cm->nfo.v;

        if (unlikely(!(cm->nfo.c.flags & CBUFM_TMEM))) goto done;
        if (unlikely(len > PAGE_SIZE)) goto done;
        ret = ((void*)(cm->nfo.c.ptr << PAGE_ORDER));
done:
        CBUF_RELEASE();
        assert(lock_contested(&cbuf_lock) != cos_get_thd_id());
        return ret;
}
```

Example: what to ignore?

```
cbuf2buf(struct cbuf cb, int len)
{
        u32_t id;
        struct cbuf_meta *cm;
        union cbufm_info ci;
        long cbidx;
        if (len) {
                int again;

                cbuf_unpack(&cb, &id);

                CBUF_TAKE();
                cbidx = cbid_to_meta_idx(id);
                do {
                        again = 0;
                        cm = cbuf_lookup_addr(cbidx);
                        if (!cm || cm->nfo.v == 0) {
                                if (cbuf_2buf_get(id, len)) {
                                        CBUF_RELEASE();
                                        return NULL
                                }
                                again = 1;
                        }
                } while (again || !cm->nfo.v);
                ci.v = cm->nfo.v;

                if ((cm->nfo.c.flags & CBUFM_TMEM)) {
                        if (len <= PAGE_SIZE) {
                                CBUF_RELEASE();
                                return ((void*)
                                        (cm->nfo.c.ptr << PAGE_ORDER));
                        }
                }
                CBUF_RELEASE();
                return NULL;
        }
        return NULL;
```

Small register file? Be *Actively* Lazy

- What can I *choose* to ignore?
  - error cases – how discriminate these conditions?
  - edge cases – what's an edge case?
  - variables – until used in an important context
  - loops – distill to one operation
- largest "productivity" boost in reading
  - *hypothesis should enable these simplifications*
- goal: code details $\rightarrow$ hypothesis
  - remember hypothesis, not details
- Note: very little of this applies to *algorithms*

Hypothesis vs. Code: DEATHMATCH

Relationship between code and hypothesis

- Hypothesis about code $\rightarrow$ interpretation of code
- Interpretation of code $\rightarrow$ confirm/refute hypothesis

Important relationship to understand

- "understanding" code is
  - making stronger, higher-level, confirmed hypothesis
  - enables a deeper investigation of code
  - *"virtuous cycle"*

Hypothesis vs. Code: DEATHMATCH

$$\text{hypothesis}_0 = f(\text{goal, behavior, design, surface questions})$$
$$\text{hypothesis}_n = f(\text{goal, behavior, design, } \text{brain}^{eval}_{apply}(\text{code, } \{\text{hypothesis}_m | \forall m < n\}))$$

Hypothesis vs. Code: DEATHMATCH

$$\text{hypothesis}_0 = f(\text{goal, behavior, design, surface questions})$$
$$\text{hypothesis}_n = f(\text{goal, behavior, design, } \text{brain}^{eval}_{apply}(\text{code, } \{\text{hypothesis}_m | \forall m < n\}))$$

$$\text{understanding} = \{\text{hypothesis}_m \mid \text{m} \leq \text{sufficiently large } n\}$$

The Algorithm

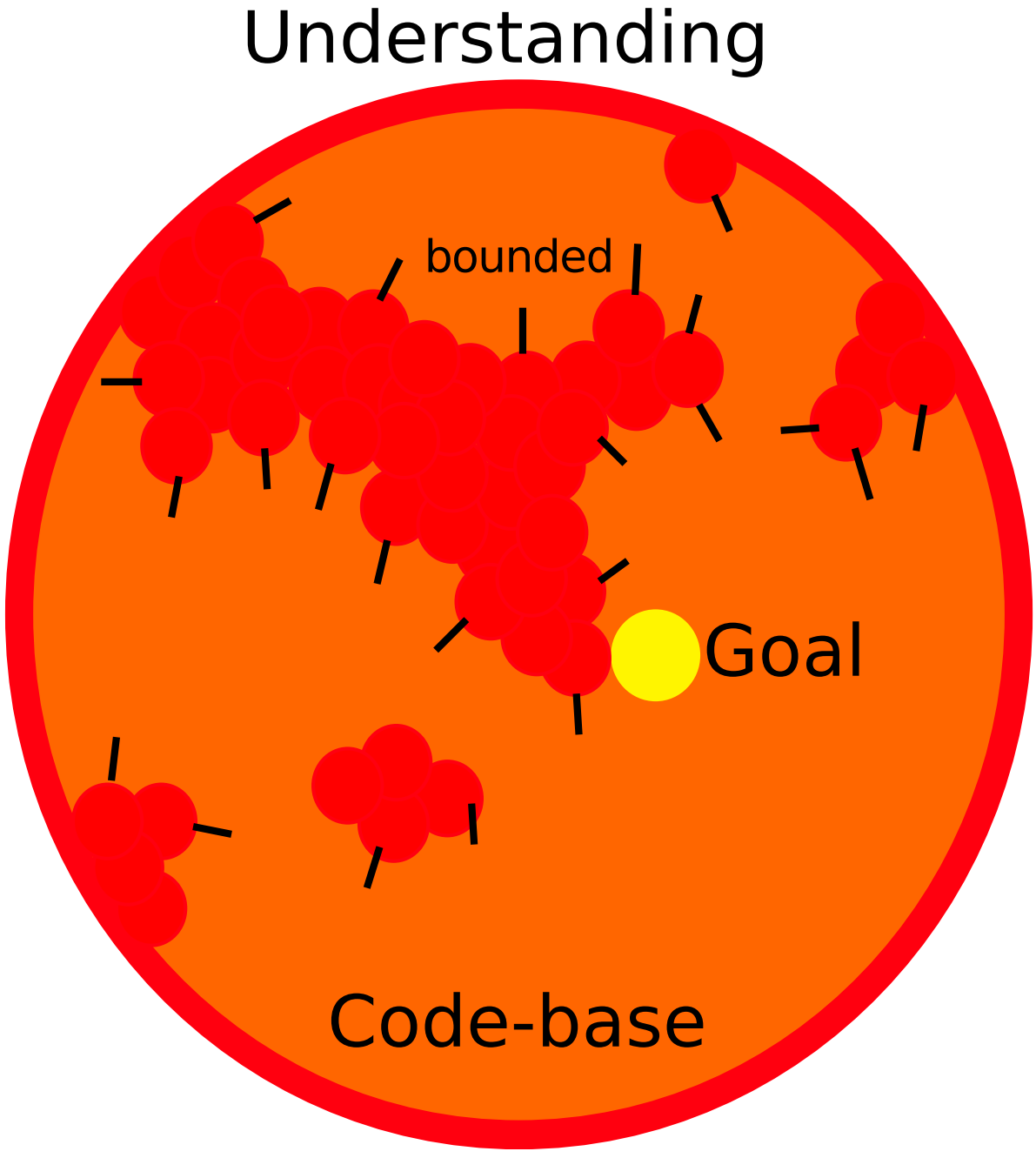
Branch and (aggressively) Bound

- Peel-First Search (PFS) through code
- aggressively ignore branch possibilities (bound)
  - error/edge cases, defined by current hypothesis
  - deeper levels of abstraction than currently useful
- refine, confirm, and synthesize hypothesis

Whole purpose: *actively engage* in code-understanding

- large difference between reading & understanding a book

Understanding Code: Bounded, PFS



Naming

Want your code to be impossible to understand?

One simple, easy step!!!

Naming

Want your code to be impossible to understand?

One simple, easy step!!!

Choose your names badly

- best cue that a branch is unimportant
- best hypothesis formation tool

bad naming = intractable code

Naming

Want your code to be impossible to understand?
One simple, easy step!!!

Choose your names badly

- best cue that a branch is unimportant
- best hypothesis formation tool

bad naming = intractable code

Spend (lots of) time thinking about naming

ROT: don't write code if you don't value naming

The Good News

This is *a lot* easier with practice

- refine, customize your PFS and hypothesis formation
- the more you read, the easier it is to do so
- ...even in *other* code-bases

The virtuous cycle is exponential

- faster formation/confirmation of hypothesis
  $\rightarrow$ faster code interpretation
- faster code interpretation
  $\rightarrow$ faster formation/confirmation of hypothesis

Anecdote: priority-aware IPIs in Linux

The Dose of Reality

"reading" code is not enough

- "how you do'in" – skimming code,
  $\rightarrow$ *will never lead to system understanding*
- looking at swimsuit models – only reading papers,
  $\rightarrow$ *will never lead to system understanding*

Only via active engagement in

- *goal- and hypothesis-driven*
- *structured code search*

Implications

You **can not** *write good code*
until you understand how to *read code*

- no sane system optimizes for writing code (`perl` aside)
- optimize for reading code

Implications

You **can not** *write good code*
until you understand how to *read code*

- no sane system optimizes for writing code (`perl` aside)
- optimize for reading code

System style and conventions are important

- rules for hypothesis formation
  - What are `i`, `j`, and `k`???
- see COMPOSITE style guide

> `daemon(` When you learn to understand code, how can you write it to ease the process? `)`

Implications II

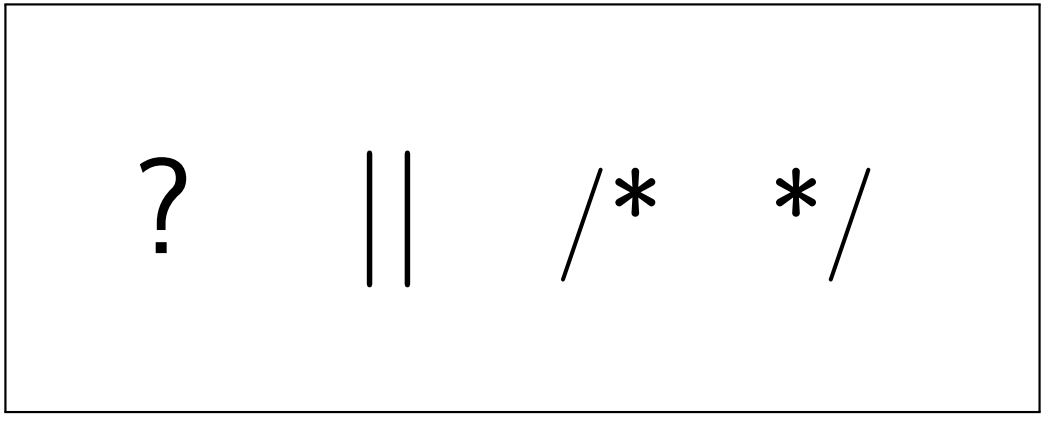
This doesn't mean code will be optimally readable

- readable code $\rightarrow\leftarrow$ optimization
  ...only optimize when necessary
- readable code $\rightarrow\leftarrow$ generic code
  ...don't over-generalize

taste in code construction

- benevolent dictator better have it

Thank You!



composite.seas.gwu.edu