```
void init_parser() {
 /* Whitespace */
 H_RULE(ws, h_in((uint8_t*)" \r\n\t", 4));
 /* Structural tokens */
 H_RULE(left_square_bracket, h_middle(h_many(ws), h_ch('['),
  h_many(ws)));
 H_RULE(left_curly_bracket, h_middle(h_many(ws), h_ch('{'),
  h_many(ws)));
 ...

 /* Forward declarations */
 HParser *value = h_indirect();
 ...
 /* Arrays */
 H_ARULE(json_array,
  h_middle(left_square_bracket,
   h_sepBy(value, comma),
  right_square_bracket));

 /* Objects */
 H_RULE(name_value_pair,
  h_sequence(json_string,
   colon, value, NULL));
 H_ARULE(json_object,
  h_middle(left_curly_bracket,
   h_sepBy(name_value_pair, comma),
  right_curly_bracket));
 h_bind_indirect(value,
  h_choice(json_object,
   json_array,
   json_number,
   json_string,
   lit_true,
   lit_false,
   lit_null,
   NULL));
```

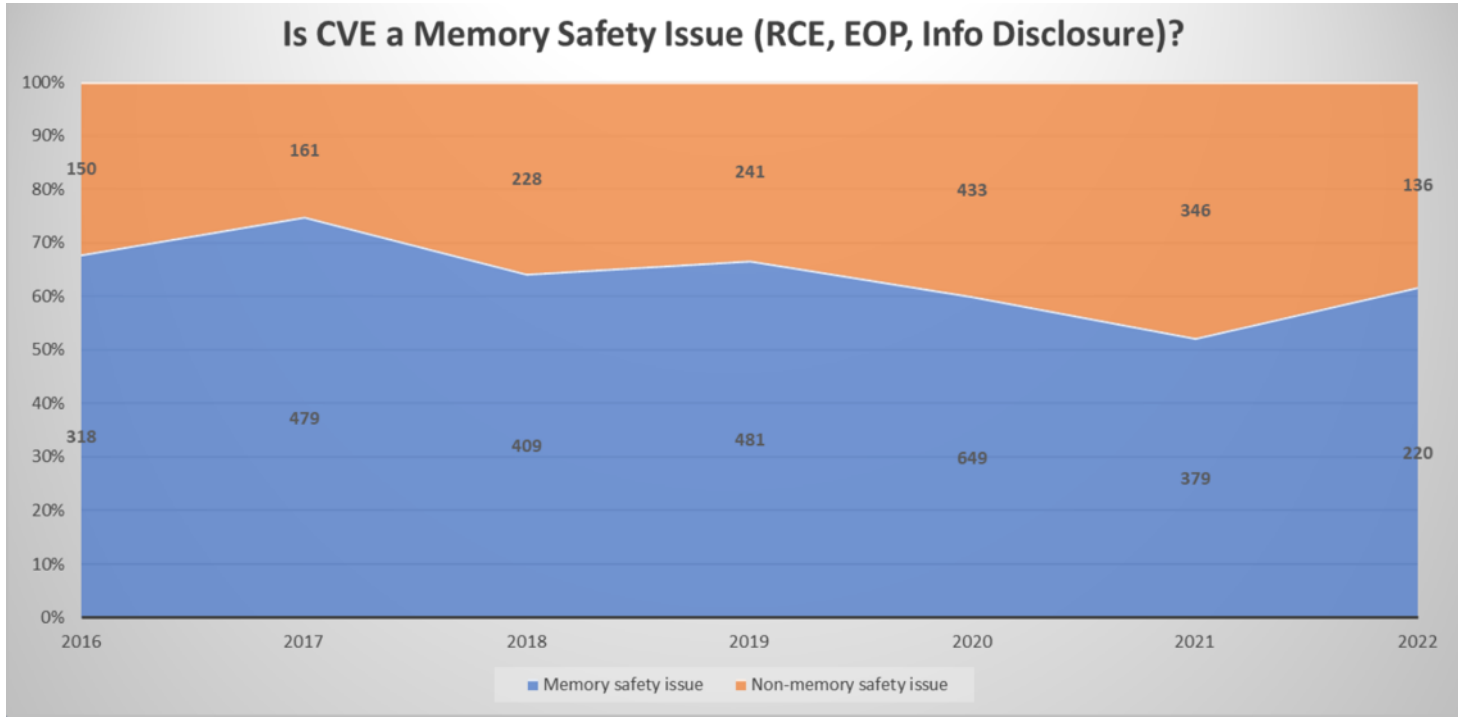
Solving the memory safety problem, once and for all

Dr. Dan Wallach
Program Manager
DARPA Information Innovation Office

Memory safety (or, the problem with programming in C)

Software written in C is not guaranteed to be *memory safe:*

- Roughly 70% of reported software vulnerabilities in critical systems (e.g., Google Chrome or Microsoft Windows) exploit memory safety flaws in C (or C++) programs[1]



Vulnerability Case Data from Microsoft (Dave Weston presentation at BlueHat IL, May 2023)

RCE = remote code execution
EOP = elevation of privilege
CVE = common vulnerability enumeration

[1]https://youtu.be/KugtcxEU6C8?si=_pA8TO1u5obD5oL5

We've tried everything!

1. developer training
2. code coverage
3. coding guidelines
4. fuzzing
5. static analysis
6. C++ subsets
7. non-executable memory
8. Control Flow Integrity (CFI)
9. Address Space Layout Randomization (ASLR)
10. sandboxing
11. hardening memory allocators
12. hardware support for the above (*)

Digression: hardware memory safety

- Sandboxing: useful to contain memory faults, limit the exploitability of a buffer overflow
  - Example: Used in the Chrome browser (and yet, security issues still happen)

- Hardware memory tagging / fat pointers
  - CHERI (fat pointers, track allocation boundaries)
  - ARM / Apple Extended Memory Tagging Extensions (shipping on M5 Macs / iPhone 17)
    - 4-bit tags on memory allocations and pointers; writes that cross tag boundaries halt the program
    - Includes hardware features to protect against speculation attacks!

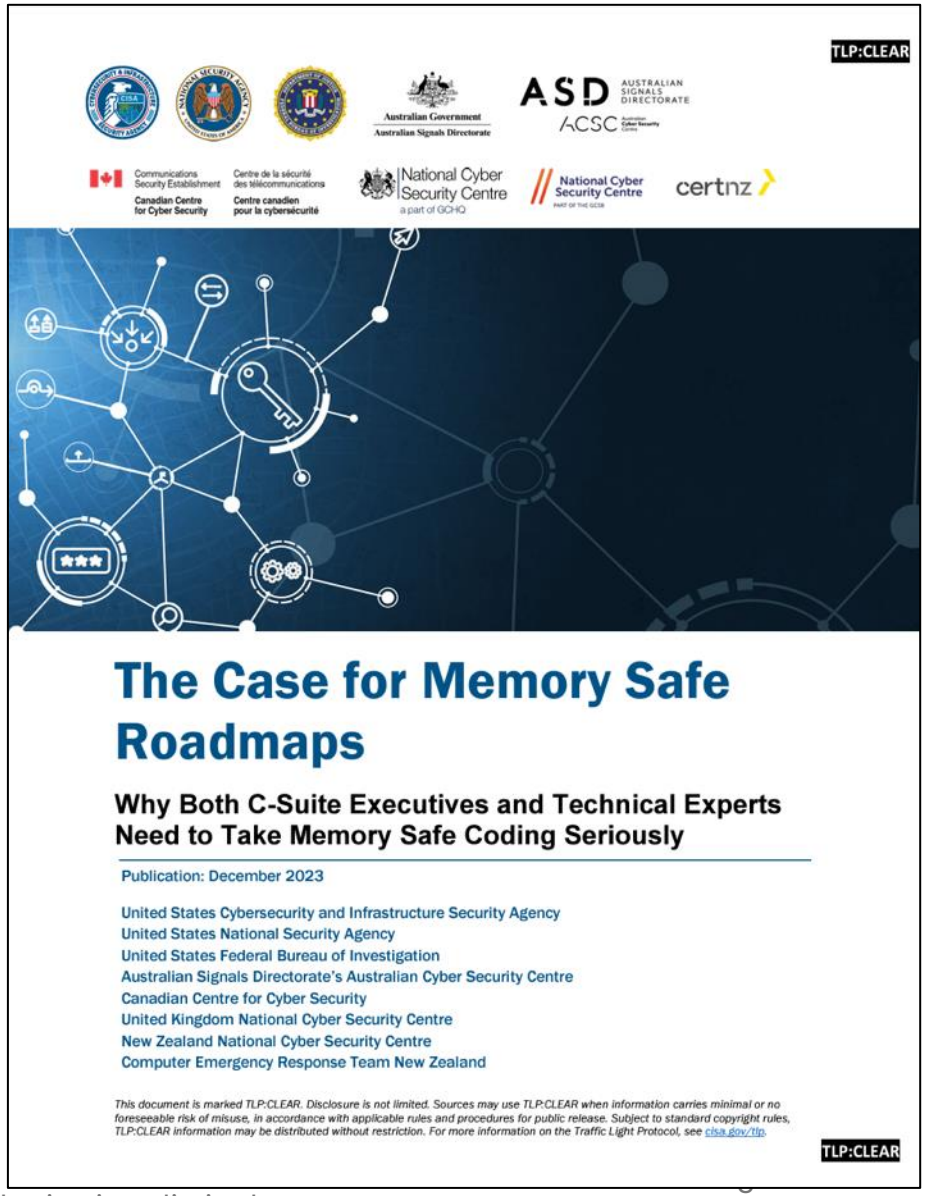
- But what do we do with all the legacy computers in the world?

CHERI = Capability Hardware Enhanced RISC Instructions

Today's advice: Incrementally rewrite in a memory safe language

"[We] encourage software manufacturers to move security considerations earlier in the [software development lifecycle], which can improve product security and reliability upon deployment. Depending on the programming language in question, there may be an increase in the amount of up-front work[1]"

- Engineering and policy challenge: rewriting legacy software is expensive
- Example: Google's experience with Android
  - Incremental rewriting of sensitive code into Rust for several years
  - Substantially fewer memory-safety vulnerabilities, but still present
- For legacy software (open source or industry), insufficient funding available (one industry estimate: $1 trillion cost to rewrite one billion lines of code)


TLP:CLEAR

Communications Security Establishment Canadian Centre for Cyber Security | Centre de la sécurité des télécommunications Centre canadien pour la cybersécurité | Australian Government Australian Signals Directorate | ASD AUSTRALIAN SIGNALS DIRECTORATE ACSC | National Cyber Security Centre a part of GCHQ | National Cyber Security Centre | certnz

The Case for Memory Safe Roadmaps

Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously

Publication: December 2023

United States Cybersecurity and Infrastructure Security Agency
United States National Security Agency
United States Federal Bureau of Investigation
Australian Signals Directorate's Australian Cyber Security Centre
Canadian Centre for Cyber Security
United Kingdom National Cyber Security Centre
New Zealand National Cyber Security Centre
Computer Emergency Response Team New Zealand

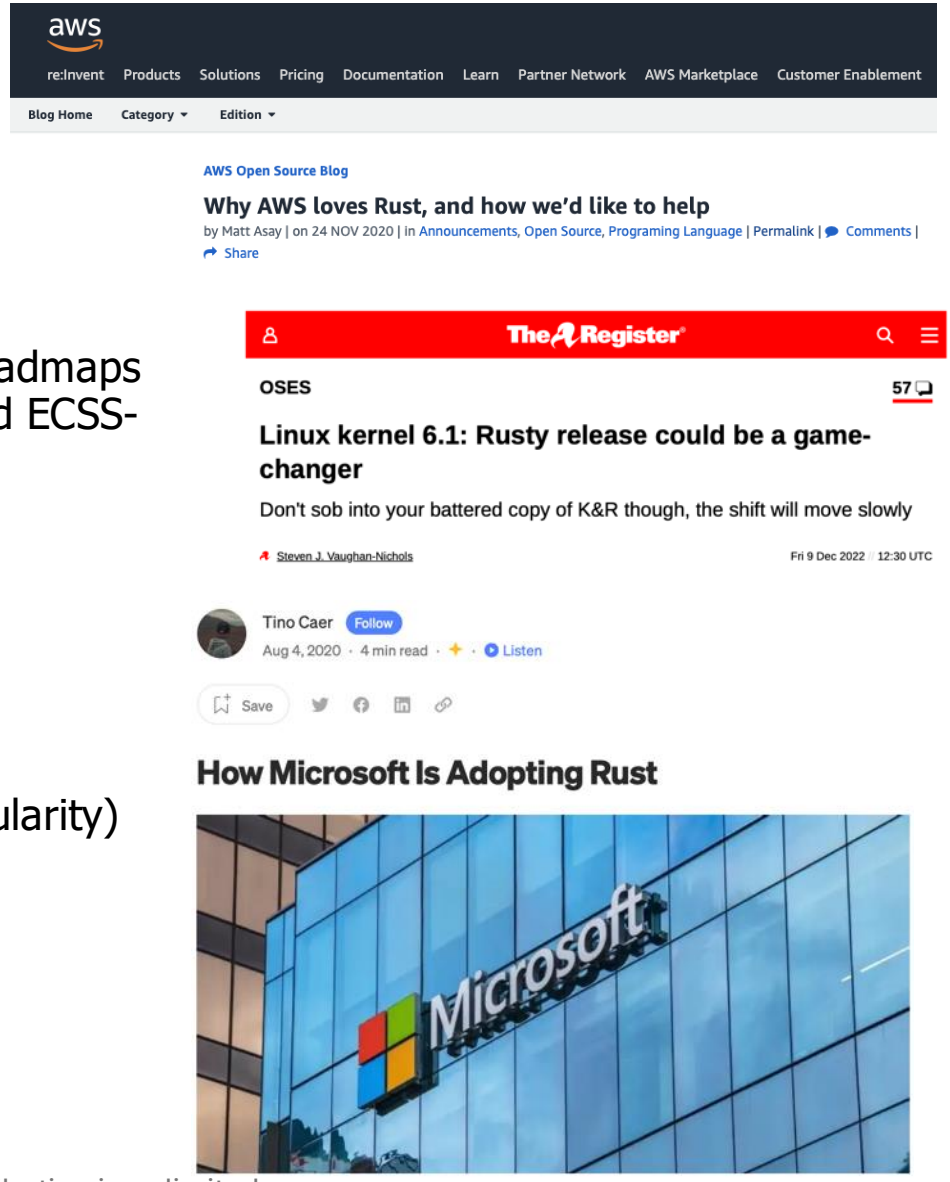
This document is marked TLP:CLEAR. Disclosure is not limited. Sources may use TLP:CLEAR when information carries minimal or no foreseeable risk of misuse, in accordance with applicable rules and procedures for public release. Subject to standard copyright rules, TLP:CLEAR information may be distributed without restriction. For more information on the Traffic Light Protocol, see cisa.gov/tlp.

TLP:CLEAR

[1]https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf

Rust: the memory-safe systems programming language of choice

- Rust programs are memory safe "by construction"
  - Guaranteed memory safety (or rejected by the compiler)
  - Competitive performance and memory use to C
  - "Unsafe" escape hatch when necessary
  - Efficient interoperation with legacy C and C++
  - Multiple vendors offer ISO 26262 "qualified" Rust (for automotive); roadmaps include avionics (DO-178), rail (EN-50128), space (ECSS-E-ST-40C and ECSS-Q-ST-80C) and others
- Growing popularity → easier to hire or train Rust programmers
- Today: full rewrite of legacy C software to Rust is too expensive
- If we could do *automatic, high quality* conversion
  - Immediate, permanently improved security posture for minimal cost
  - Ongoing benefits (e.g., cheaper maintenance due to better code modularity)
  - Open source and defense performers would be enthusiastic adopters



Rust

If we eliminate memory unsafety in software, do we need hardware?

- If we somehow rewrote every line of C and C++ into safe Rust, we'd eliminate:
  - Buffer overflows ("spatial memory safety")
  - Use after free / double free ("temporal memory safety")
  - Type confusion
  - And also have an impact on null pointer dereference (*), concurrency bugs, and integer overflow/underflow

- But we wouldn't solve microarchitectural timing attacks (Meltdown, Spectre, etc.)

- Thought experiment: do memory safe languages know enough to stop undesirable speculation?
  - Example: How do we squash a speculative load beyond the bounds of an array?
  - Rust arrays know their length
  - Hypothetical "bounded load instruction": take source address register, offset, bound, and destination register
    - Or, could use fat pointers, tagged memory, etc.
  - Squash any memory speculation beyond the specified bound

(*) The Rust equivalent: calling `unwrap()` on an empty Option. This led to a Cloudflare outage in Nov '25.

Current state of the art (not all translations are equally good)

Original C code

```c
int main(int argc, char **argv) {
    printf("Hello, %s\n", argv[1]);
}
```

Safe, idiomatic translation (hand-written)

```rust
fn main() {
    let name = std::env::args().nth(1)
      .unwrap_or(&String::from(""));

    println!("Hello, {}", name);
}
```

Supplies empty string if missing arguments

Bug-for-bug compatible (e.g., will crash if args are missing)

Unsafe Rust translation (via c2rust)

```rust
#![allow(dead_code, mutable_transmutes, non_camel_case_types, non_snake_case,
non_upper_case_globals, unused_assignments, unused_mut)]
#![register_tool(c2rust)]
#![feature(register_tool)]
extern "C" {
    fn printf(_: *const libc::c_char, _: ...) -> libc::c_int;
}
unsafe fn main_0(
    mut argc: libc::c_int,
    mut argv: *mut *mut libc::c_char,
) -> libc::c_int {
    printf(
        b"Hello, %s\n\0" as *const u8 as *const libc::c_char,
        *argv.offset(1 as libc::c_int as isize),
    );
    return 0;
}
pub fn main() {
    let mut args: Vec::<*mut libc::c_char> = Vec::new();
    for arg in ::std::env::args() {
        args.push(
            (::std::ffi::CString::new(arg))
                .expect("Failed to convert argument into CString.")
                .into_raw(),
        );
    }
    args.push(::std::ptr::null_mut());
    unsafe {
        ::std::process::exit(
            main_0(
                (args.len() - 1) as libc::c_int,
                args.as_mut_ptr() as *mut *mut libc::c_char,
            ) as i32,
        )
    }
}
```

LLMs can translate code, sometimes very well

- "Please translate this C to safe, idiomatic, simple Rust" and it did the job perfectly
  - It even renamed the buffer to "password" and correctly handled missing arguments

Original C code

```c
int main(int argc, char **argv) {
    char buff[8];
    int success = 0;

    strcpy(buff, argv[1]);

    if(!strcmp(buff, "s3cr8tpw")) {
        success = 1;
    }

    if(success) {
        printf("Welcome!\n");
    }
}
```

Long inputs will overwrite the success flag.

Rust translation (by Google Gemini)

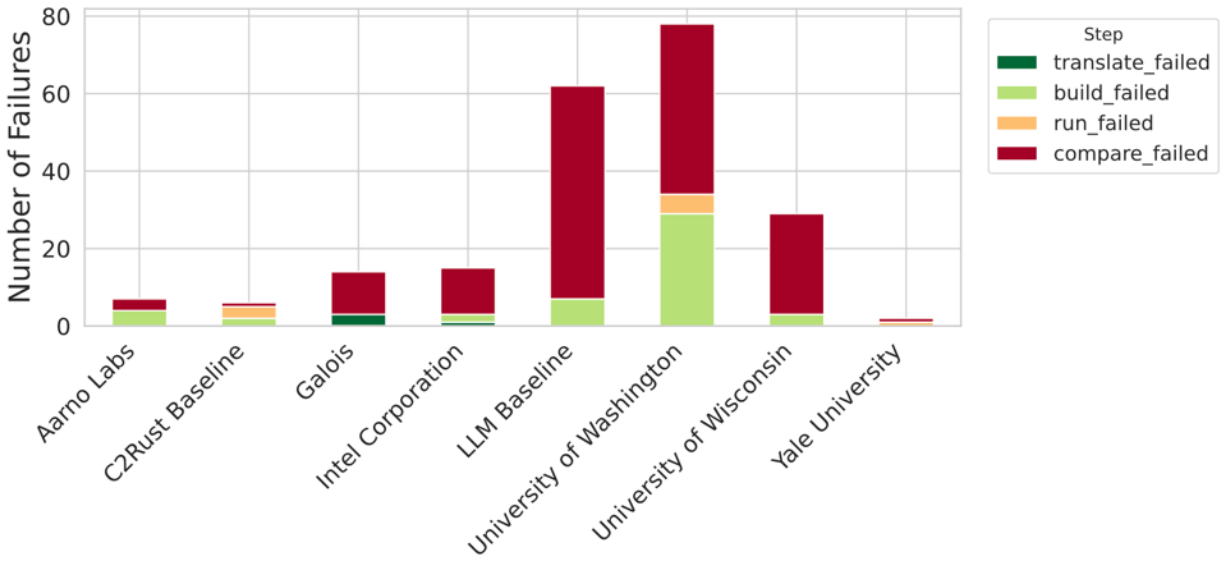
```rust
fn main() {
    let password = if let Some(arg) = args().nth(1) {
        arg
    } else {
        return; // Exit program if no argument provided
    };

    if password == "s3cr8tpw" {
        println!("Welcome!");
    }
}
```

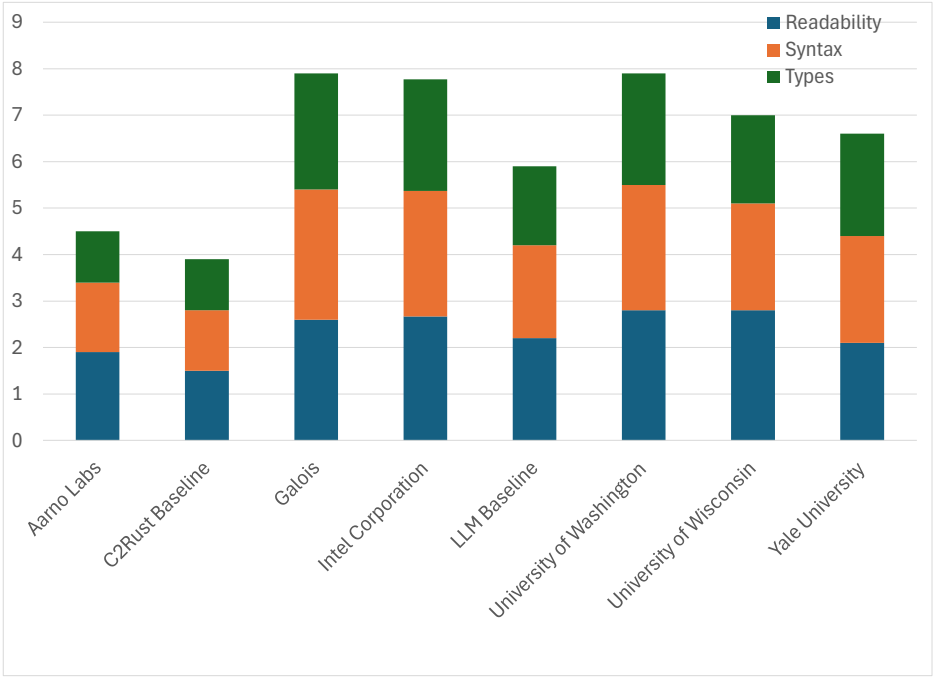
TRACTOR progress report

- TRACTOR started rolling in June 2025
  - Six teams, taking *very* different approaches (from ML-centric through PL-centric)
  - Initial progress? We evaluated the teams on a number of "microbenchmarks" of purpose-written C with or without various pathologies. Too early to declare anybody "winning" or "losing".

Translation failures (lower is better)



Idiomatic code quality (higher is better)



https://www.darpa.mil/research/programs/translating-all-c-to-rust

LLMs are increasingly amazing; do we still need TRACTOR?

- Code translation is hard enough; evidence of code equivalence is harder still

- Legacy codebases often lack sophisticated test suites
  - If the code is the spec, then what does it means for a translation to be “equivalent”?

- TRACTOR ultimately needs to convince a team to abandon its old C code and move to a new Rust codebase.

  This requires evidence.
  It’s a high bar.

https://www.anthropic.com/engineering/building-c-compiler

While we can’t do everything automatically, what should be the top priority?

Data format complexities lead to vulnerabilities in parsers

"PDF is the Most Common Malware Vector."
- Schneider on Security

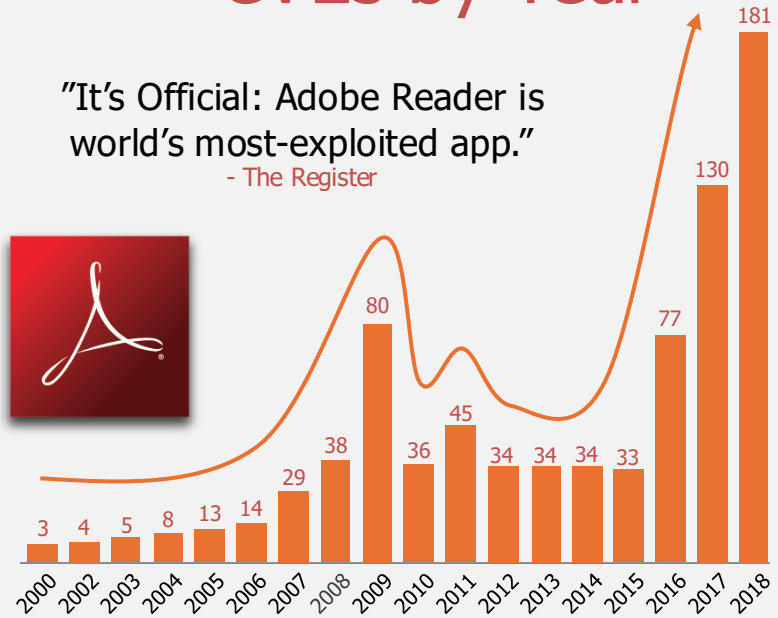
"One of the biggest reasons that PDF exploits blossomed in 2009 was Adobe Reader's ubiquity."
- Roul Schouwenberg, Kaspersky Researcher

"Researchers analyzed emails and identified 300,000 unique malicious documents, representing 48% of all malicious files detected in the last 12 months."
- Barracuda Networks

"In all of last year, our Capture ATP sandbox discovered more than 47,000 new attack variants in PDF files."
- Capture ATP Sandbox

PDF-Related CVEs by Year

"It's Official: Adobe Reader is world's most-exploited app."
- The Register



"Data Hiding: The PDF syntax is very loose and could be used to hide information easily."
- DSTO-TR-2730 (Unclassified)

"The Pdf file format specification and the properties of the viewer allow malware authors a significant degree of freedom when designing and developing a threat."
- Symantec Security Response

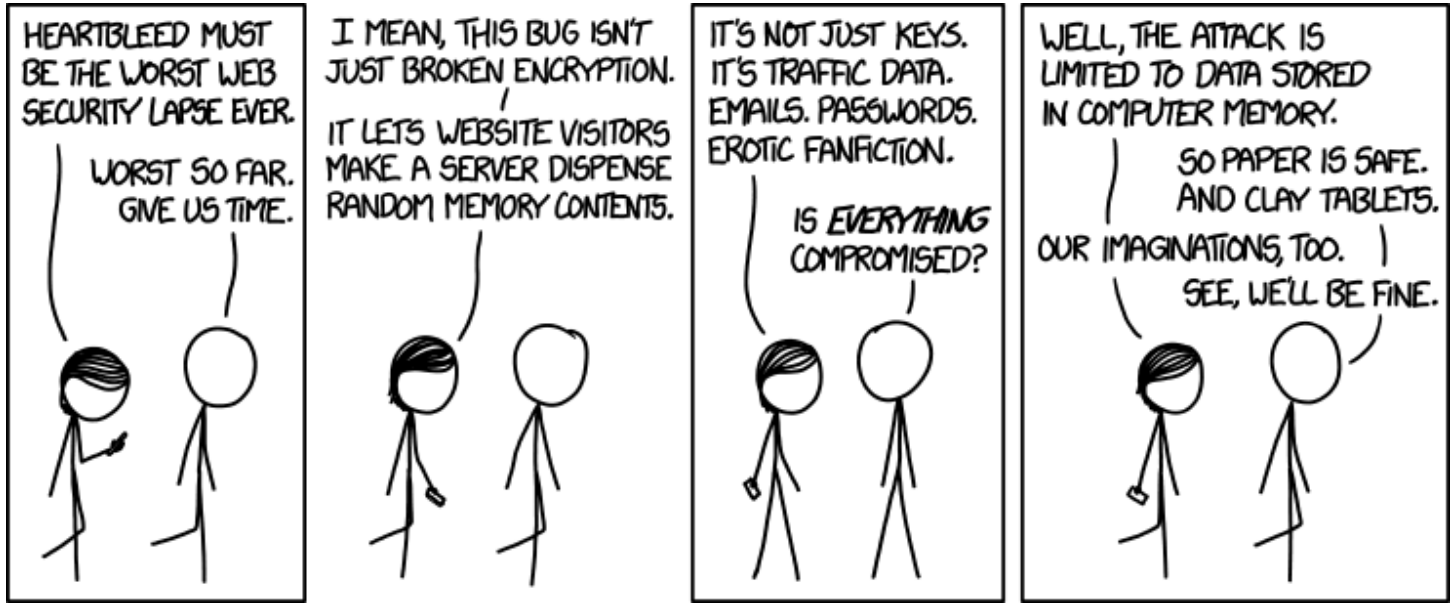
" As with Microsoft Office documents in the past, the PDF file format has become a target for malware authors and is currently being widely exploited as a means to deposit malware onto computers. "
- Symantec Security Response

"Recent attacks like SpicyOmelette illustrate a typical Cobalt Group attack pattern starting with a spear phishing email containing a malicious PDF file."
- Cylance 2019 Threat Report

Parsers: The code that read these complex data formats

Parsers (computer security perspective)

- Parsers convert (untrusted) bytes to (sensitive) internal data structures
- Parsers are the outer edge of the attack surface of every program!
- And, in C or C++, hand-written parsers are (arguably) the source of 80% of CVEs
  - Developers take shortcuts, make unsafe assumptions
  - Example: Heartbleed OpenSSL bug (2014), trusting a length field allows an attacker to read sensitive memory



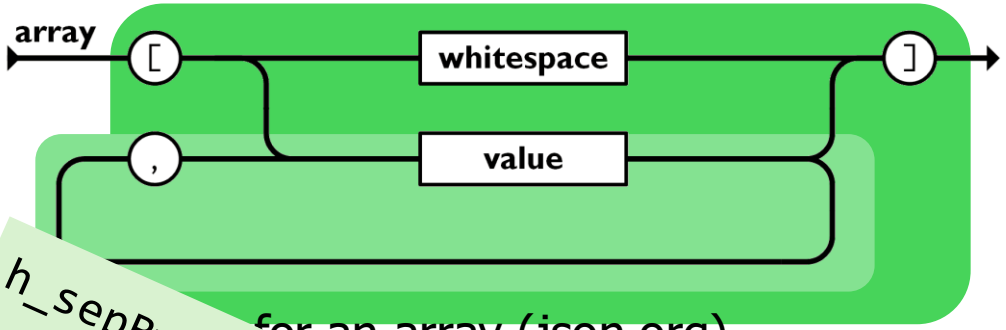
https://xkcd.com/1353/

CVE: Common Vulnerabilities and Exposures (https://www.cve.org/About/Overview)

Parsers (functional programming nerd perspective)

Parsers are *algebraic data types!* We can *combine* small parsers into bigger parsers

Example JSON parser (written with the Hammer parser combinator library for C)



for an array (json.org)

```
void init_parser() {
  /* Whitespace */
  H_RULE(ws, h_in((uint8_t*)" \r\n\t", 4));
  /* Structural tokens */
  H_RULE(left_square_bracket,
    h_middle(h_many(ws), h_ch('['), h_many(ws)));
  H_RULE(right_square_bracket,
    h_middle(h_many(ws), h_ch(']'), h_many(ws)));
  H_RULE(comma, h_middle(h_many(ws), h_ch(','), h_many(ws)));

…

  /* Forward declarations */
  HParser *value = h_indirect();

…

  /* Arrays */
  H_ARULE(json_array,
    h_middle(left_square_bracket,
      h_sepBy(value, comma),
      right_square_bracket));
```

h_sepBy (separate /

h_

t

Notably absent: error-handling code. (But it's still there under the hood.)

Parser combinator code looks (more) like an abstract spec.

https://reasonml.chat/t/a-gentle-introduction-to-parser-combinators-and-angstrom/2543
(Tutorial for these concepts using the Angstrom library for TypeScript)
Source: https://github.com/sergeybratus/HammerPrimer/blob/master/lecture_13/json.c

How hard is it to write a parser?

CVEs for JSON parsers in C and C++

- "Find all the CVEs for popular C and C++ JSON parsers and categorize the vulnerabilities." (Gemini 2.0 Flash, "Deep Research", May 2025. "Gemini can make mistakes, so double-check it")

Image credit: https://vlipsy.com/vlip/spongebob-squarepants-a-little-longer-than-a-few-minutes-later-pTepukCR

Lexical analysis / tokenization vulnerabilities

CVE ID	Affected Library	Description	Severity
CVE-2016-4303	cJSON	Mishandling of UTF8/16 strings in parse_string leading to heap-based buffer overflow	Unknown
CVE-2016-10749	cJSON	Buffer over-read in parse_string when string ends with backslash	Unknown

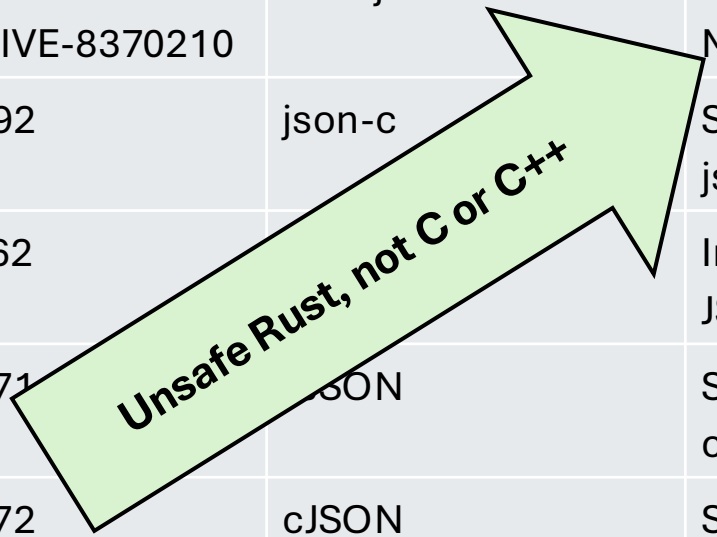
Data handling and semantic interpretation vulnerabilities

CVE ID	Affected Library	Description	Severity
CVE-2024-38517	RapidJSON	Integer underflow in GenericReader::ParseNumber()	High
CVE-2024-39684	RapidJSON	Integer overflow in GenericReader::ParseNumber()	High
CVE-2023-26819	cJSON	Denial of service via crafted JSON document with a large number	Low

A sampling of parser CVEs for JSON in C and C++

Memory management vulnerabilities

CVE ID	Affected Library	Description	Severity
SNYK-UNMANAGED-NLOHMANNJSON-6387367	nlohmann/json	Heap-based buffer overflow during CBOR parsing due to unclosed UTF-8 string	High
CVE-2019-15550	simdjson	Out-of-bounds read and incorrect crossing of a page boundary	High
SNYK-RUST-SIMDJSONDERIVE-8370210	simdjson-derive	Access of uninitialized pointer due to misuse of MaybeUninit (Rust)	High
CVE-2021-32292	json-c	Stack-buffer-overflow in parseit function of json_parse sample program	Critical
CVE-2020-12762		Integer overflow and out-of-bounds write via large JSON file	Unknown
CVE-2023-50471	cJSON	Segmentation violation via cJSON_InsertItemInArray	High
CVE-2023-50472	cJSON	Segmentation violation via cJSON_SetValuestring	Unknown
CVE-2024-31755	cJSON	Segmentation violation via cJSON_SetValuestring with NULL argument	Unknown



Error handling and input validation vulnerabilities

CVE ID	Affected Library	Description	Severity
CVE-2024-38525	dd-trace-cpp (using nlohmann/json)	Uncaught exception when logging malformed unicode	High
CVE-2024-34363	Envoy (using nlohmann/json)	Uncaught exception when serializing incomplete UTF-8 strings	High
CVE-2019-1010239	cJSON	Null dereference in cJSON_GetObjectItemCaseSensitive( ) due to improper condition check	High
AIKIDO-2024-10263	JsonCpp	Out-of-bounds read in getLocationLineAndColumn during error message generation	Low

A sampling of parser CVEs for JSON in C and C++

Denial of service

CVE ID	Affected Library	Description	Severity
CVE-2013-6401	Jansson	Predictable hash collisions leading to denial of service	Medium

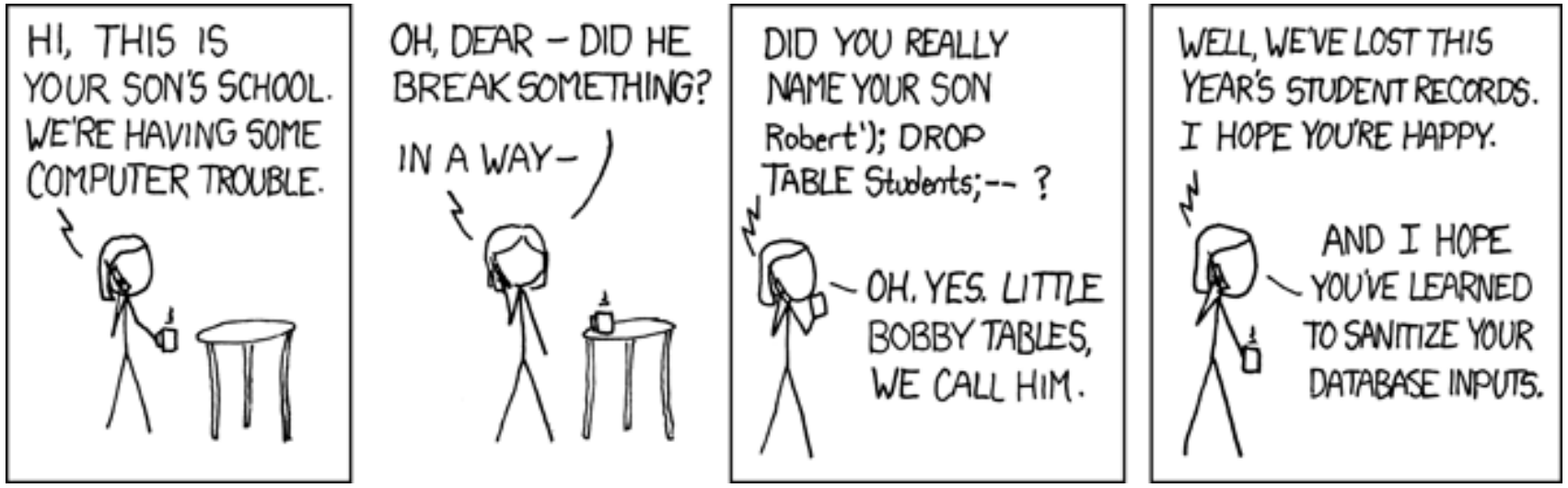
Let's use a safe programming language!

Memory-safe programming languages (roughly) guarantee code will have well-defined behavior

- Memory safety would have defeated the Heartbleed vulnerability (and many, many others)

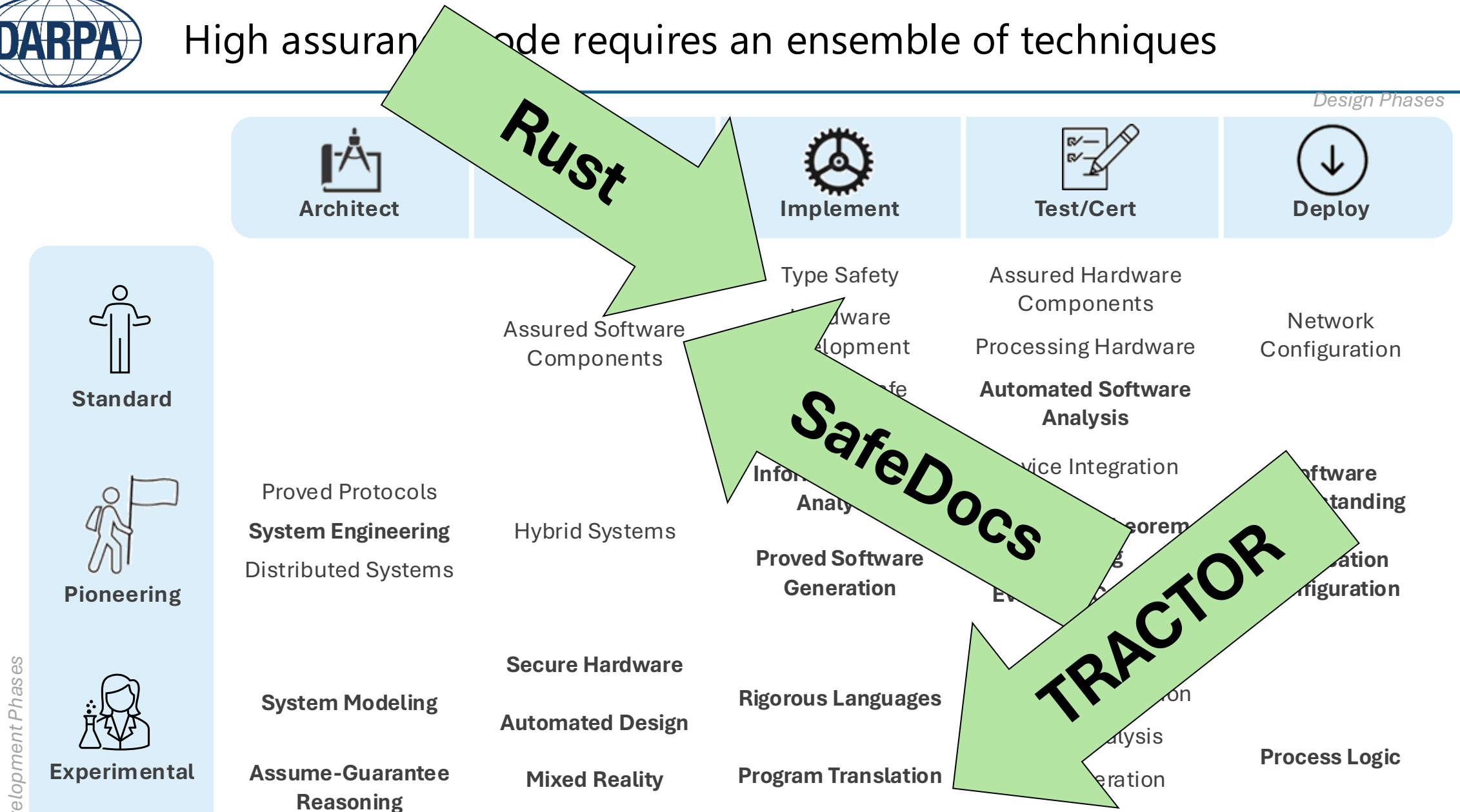
A buggy parser, even in a safe language, can still be bad for security

- Security decisions are made based on the outputs of parsers
- Code injection attacks (cross-site scripting, SQL injection) can be viewed as attacks on parsers



https://xkcd.com/327/

High assuran[illegible]ode requires an ensemble of techniques



DARPA: HOWTO

The DARPA Business Model

- "We've got this great research idea, we'd like to do X and Y. Will you fund it?"
  - This isn't (typically) how DARPA works
- How it actually works (simplified)

December 2022: "I have an idea to replace the C programming language."

March 2023: "When can you start?"

(time passes)

July 2024: TRACTOR is announced

November 2024: Proposals due

June 2025: TRACTOR program kickoff

DARPA initiates a new program to automate the translation of the world's highly vulnerable legacy C code to the inherently safer Rust programming language

Jul 31, 2024

Ask me anything!

- How do I learn about new DARPA programs?
  - You can sign up for email announcements (or RSS feed)
  - Information Innovation Office (I2O): https://www.darpa.mil/about/offices/i2o

- Or, you can try to learn about them before they're announced

- "Hey Dan, what are you working on next?"
  - I'm free to talk about future ideas (important caveat: "it's just a concept")
  - I'm limited once the solicitation is out (posting FAQs for everybody)

Ask me anything!

- Can I apply to an existing DARPA program that's already active?
  - Once winners are selected, the budget is allocated
- "What about programs for small business?"
  - DARPA has a small business office (SBO) that manages SBIR and STTR funding
- How do "Young Faculty Awards" work?
  - Solicitation published every year (December-ish), details vary every time
  - https://www.darpa.mil/work-with-us/communities/academia/young-faculty-award

Ask me anything!

- "What about seedlings?"
  - Seedlings, typically under $1M, are often used to explore ideas prior to a larger program
  - Common approach: discuss with a program manager before submitting

- How does the "Open Office BAA" work?
  - Catch-all for submissions that don't fit elsewhere
  - Common approach: discuss with a program manager before submitting

- Do I need to be a U.S. citizen / at a U.S. institution to be a DARPA performer?
  - Every program has different requirements; many performers are international
  - Common approach: subcontracting / collaborating with a U.S. institution

Ask me anything!

- Do universities have an advantage or disadvantage?
  - Strengths of the proposal and strengths of the team are what matters
  - Important question every PM is thinking: “can they hit the ground running?”
  - DARPA contracts have deliverables and deadlines; low performers might be cut
  - Common approach: industry and academic joint teams

- Is there anything else I should know about?
  - DARPA ERIS (Expedited Research Innovation System) Marketplace
    - Identify a “topic area”, post a 7 minute video
    - Introduce yourself and your capabilities; don’t just pitch a research effort
    - Once accepted, a PM (or anybody else in the military) can rapidly negotiate a contract
    - New “topic areas” will appear over time

- Loads more detail: https://www.darpaconnect.us/

Should I apply to be a program manager?

- Yes! DARPA PM jobs are temporary (~4 years), so we're always hiring
  - All you need is a great idea for a DARPA program

- IPA mechanism: I'm still a tenured full professor at Rice University

- Are there any restrictions?
  - You need to be a U.S. citizen
  - You need to live in the D.C. area (DARPA reimburses some costs)
  - Common approach: keep your house, rent a local apartment

Should I apply to be a program manager?

- Fun and satisfying: Learn all about the military; solve problems for everyone

SPANGDAHLEM AIR BASE, RHEINLAND-PFALZ, GERMANY
January 2026, photo by Photo by Airman 1st Class Gretchen McCarty, 52nd Fighter Wing
https://www.dvidshub.net/image/9505876/darpa-scientists-visit-spangdahlem-aviators-support-teams-looking-turn-operational-insight-into-future-defense-innovation