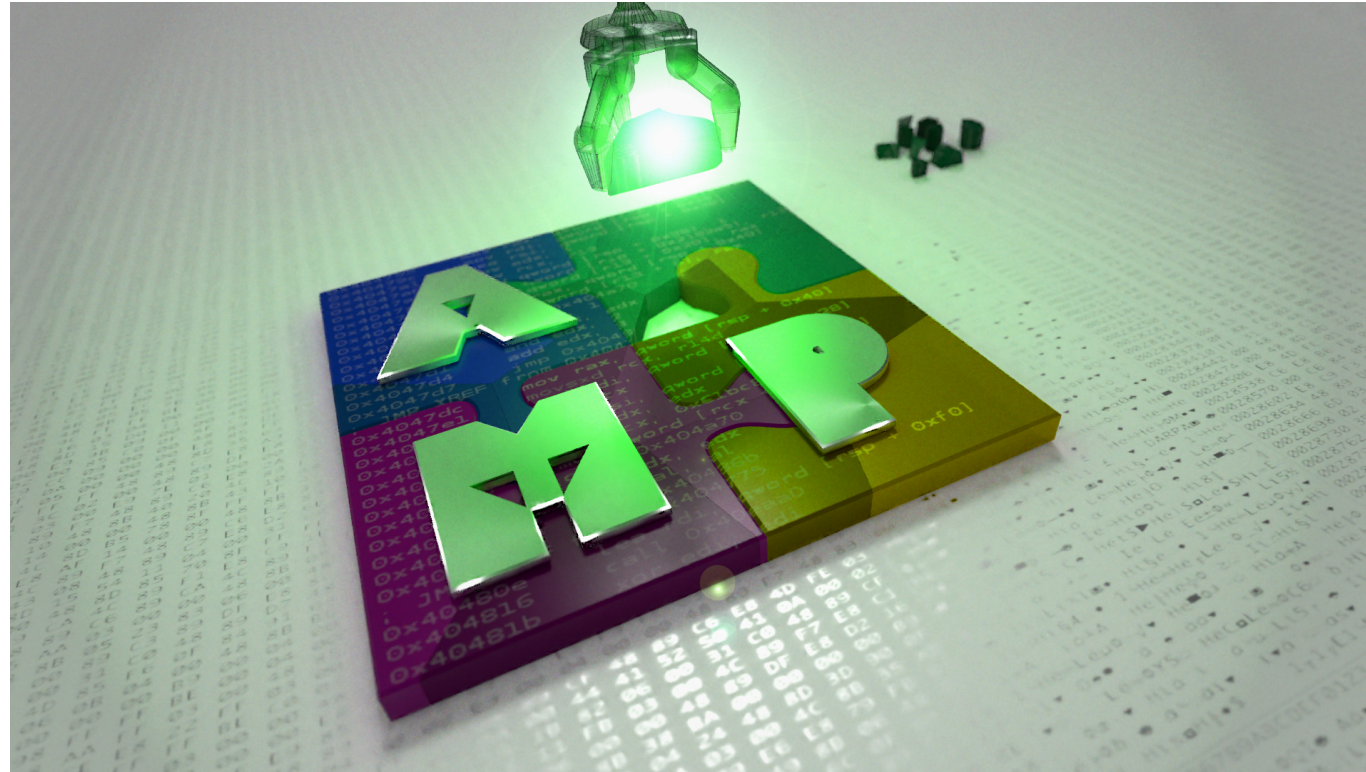


# New kinds of tools for maintaining and sustaining software and firmware

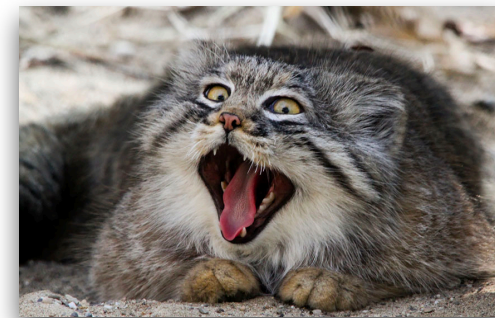
Sergey Bratus



Images of specific products throughout this presentation are used for illustrative purposes only. Use of these images is not meant to imply either endorsement or vulnerability of a product or company.

Source: Robinson, September 2019

# Sergey's disclaimers



- Obligatory: **Any opinions are mine alone and don't represent any of my employers past or present.**
- Substantive:
  - This is a personal perspective on other people's amazing work. All credit goes to them, not me.
  - This is a tiny, biased sample of a great domain. Please tell me what I am missing!
- Trivial: I am a former mathematician. I tend to see math everywhere :)



## Official Disclaimer

---

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 and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

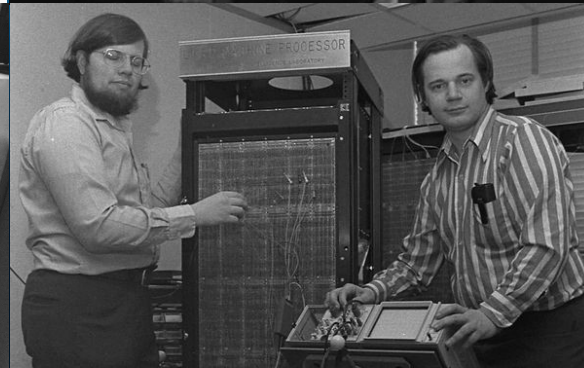
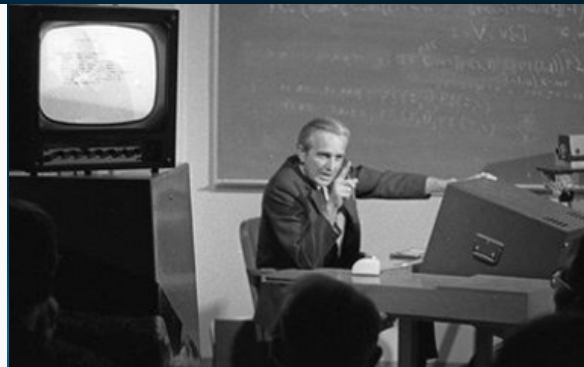
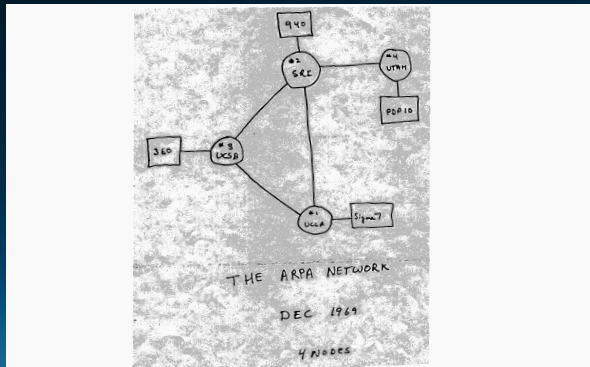
# DARPA: High Risk, High Reward Research

The Internet

Mother of All Demos

Personalized Assistant That Learns

Cyber Grand Challenge



Project MAC  
(Mathematics and Computation)

Lisp  
Processor

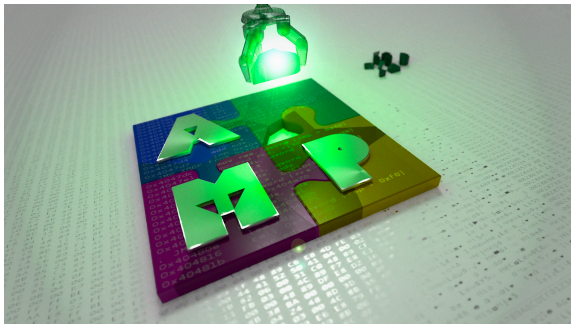
Autonomous  
Vehicles

AI and Cyber Challenge

# My DARPA programs



**Safe Documents:** Regain trust in electronic documents by creating tools to build **machine-readable unambiguous format definitions** and secure **verified parsers**



**Assured Micropatching:** Create tools for **rapid binary patching** of legacy mission-critical systems, even where the original source code or build process aren't available



**Verified Security & Performance Enhancement of Large Legacy Software:** Create practical tools for **incremental enhancement of software** systems with new verified code that is both correct-by-construction and safely composable with the rest of the system



# My DARPA programs



## **Hardening Development Toolchains Against Emergent Execution Engines:**

Develop practical tools to anticipate, isolate, and mitigate **emergent behaviors** throughout the software lifecycle, to improve security outcomes in software for complex integrated systems

E-BOSS

## **Enhanced SBOM for Optimized Software Sustainment:**

Develop Enhanced Software Bill of Material (eSBOM) advanced **metadata** technology to enable rapid triage-and-remediation of vulnerabilities in software at scale.



# We are still living out the 1960s software development revolution

---

## Awesome:

- High-level programming languages
- Automating software composition (linkers)
- Large reusable code libraries

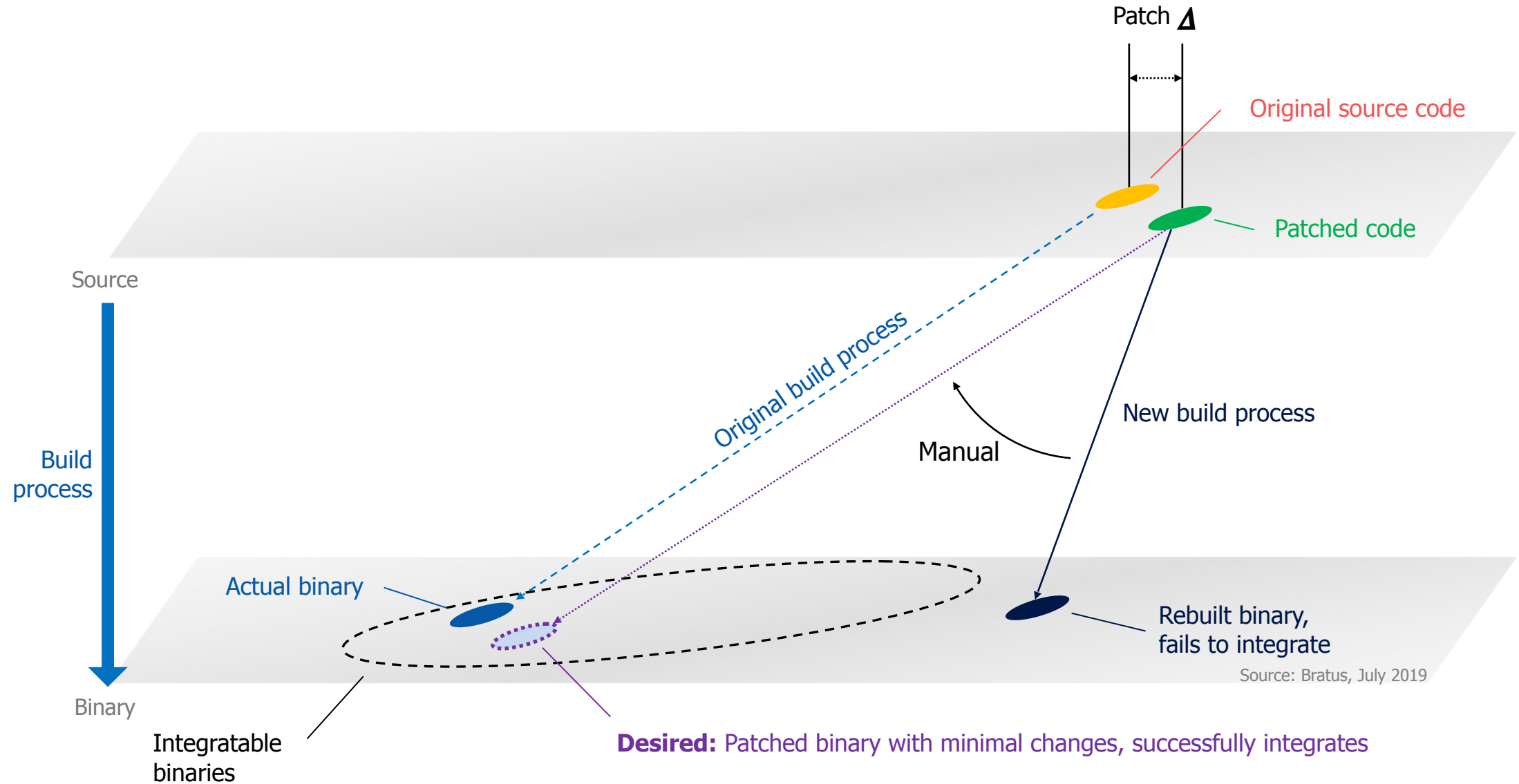
## And yet:

- Source -> compiler -> linker -> **unmaintainable** binary
- Binaries aren't meant to be incrementally updated
  - "Tear down & rebuild the house to remodel a room"

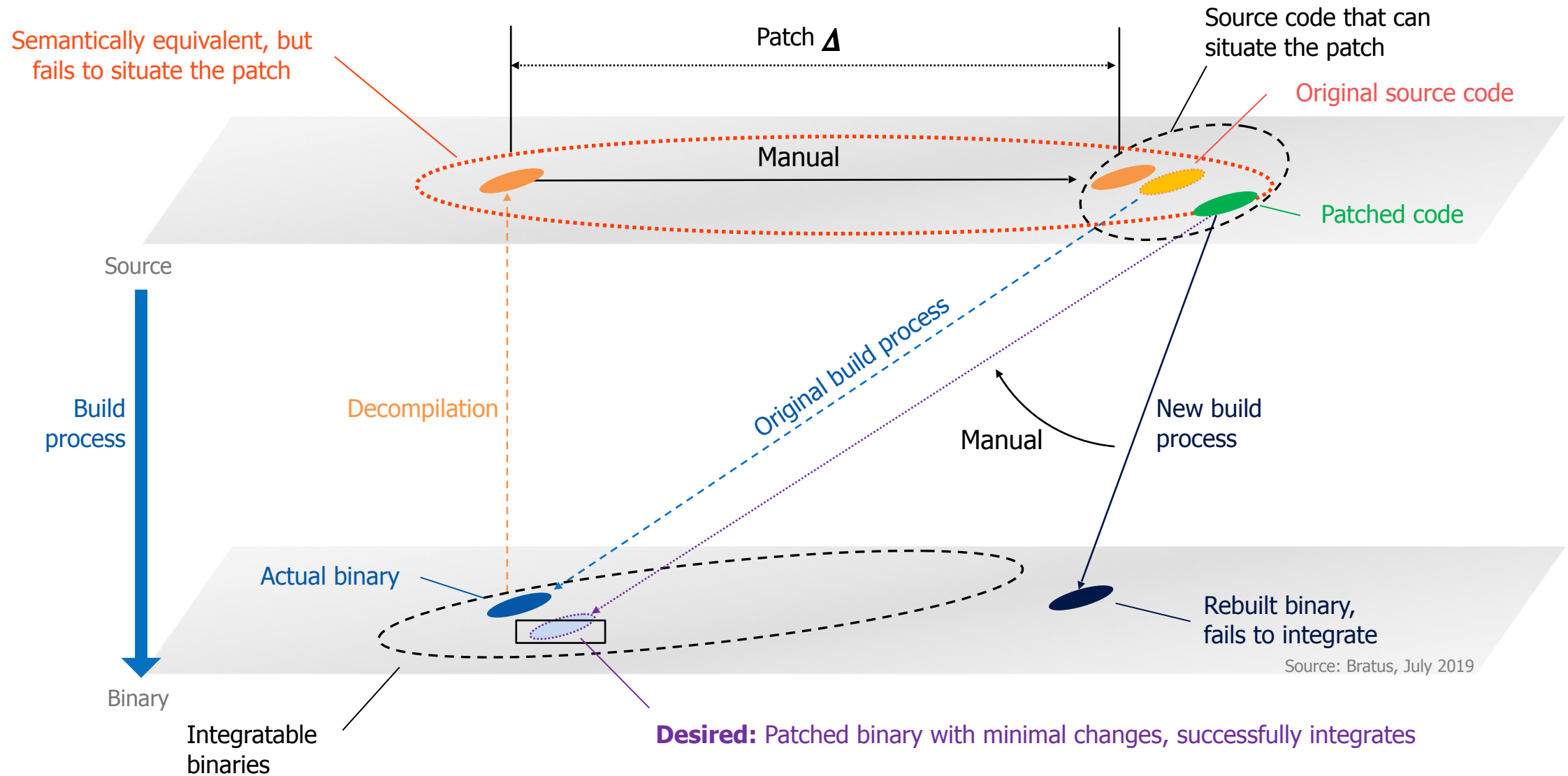


[https://en.wikipedia.org/wiki/Grace\\_Hopper](https://en.wikipedia.org/wiki/Grace_Hopper)

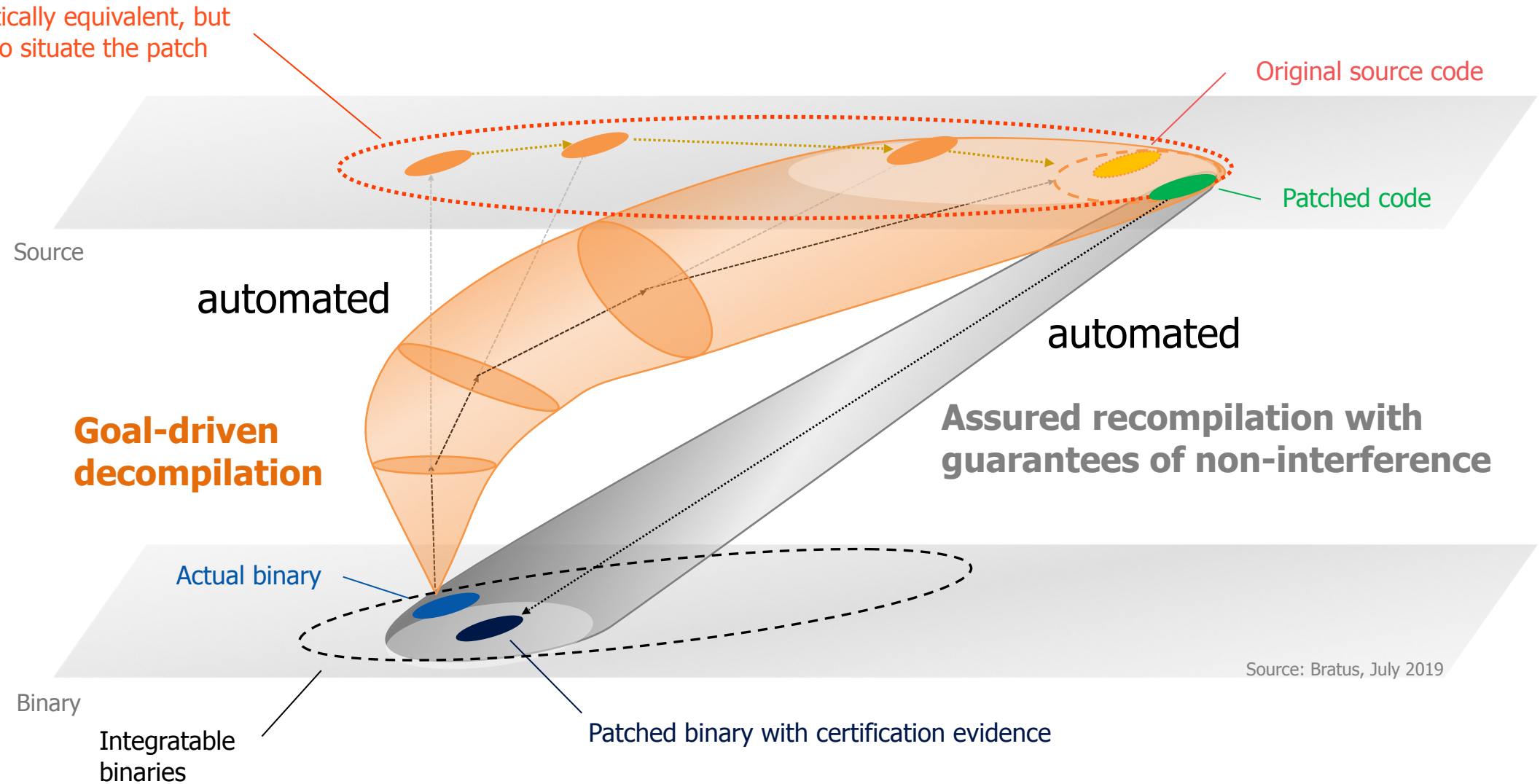
# Challenge: Rebuilding and re-integration needs costly manual effort



# Challenge: Rebuilding and re-integration needs costly manual effort

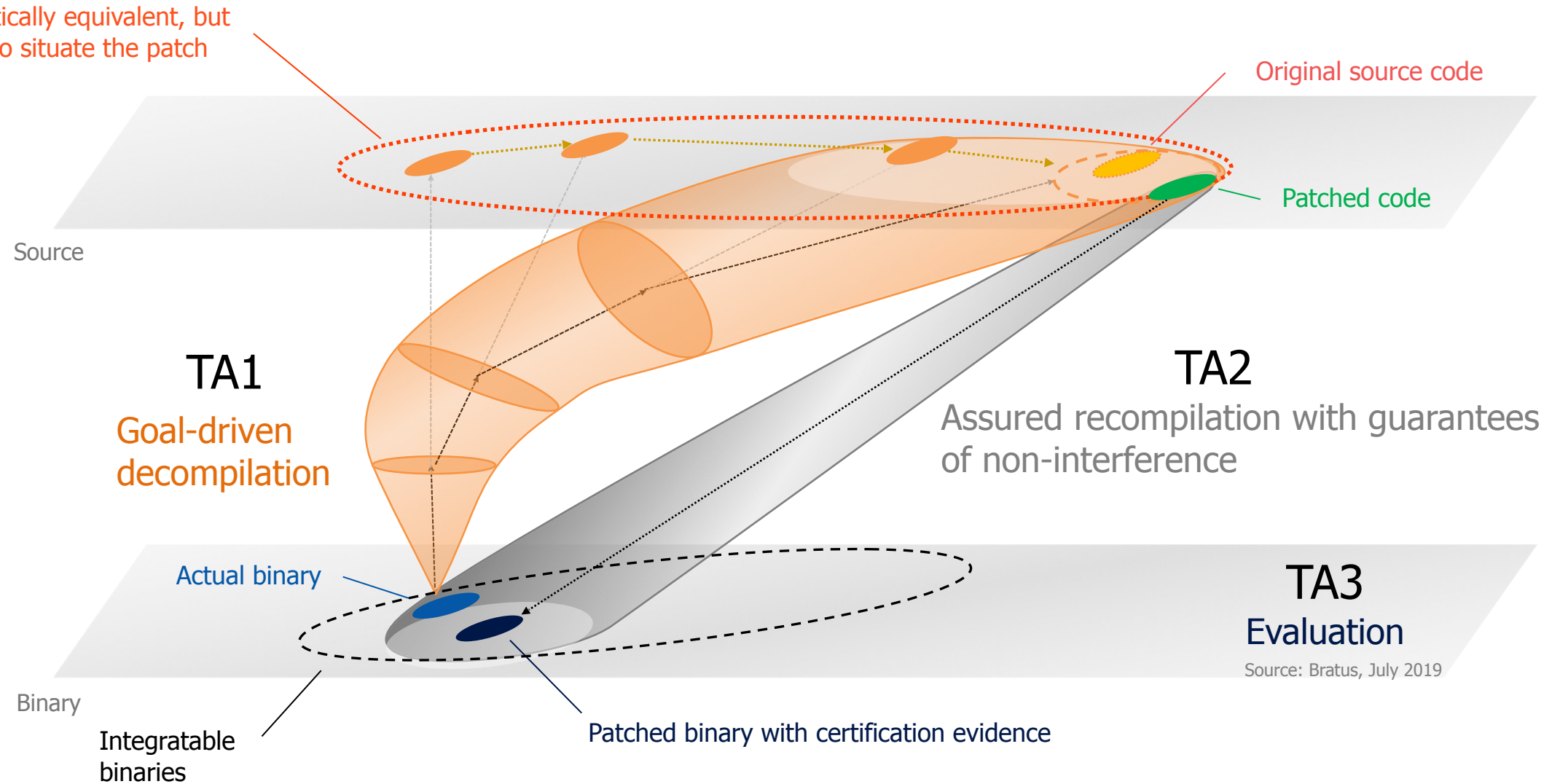


# Vision: Micropatches with certification evidence via automation



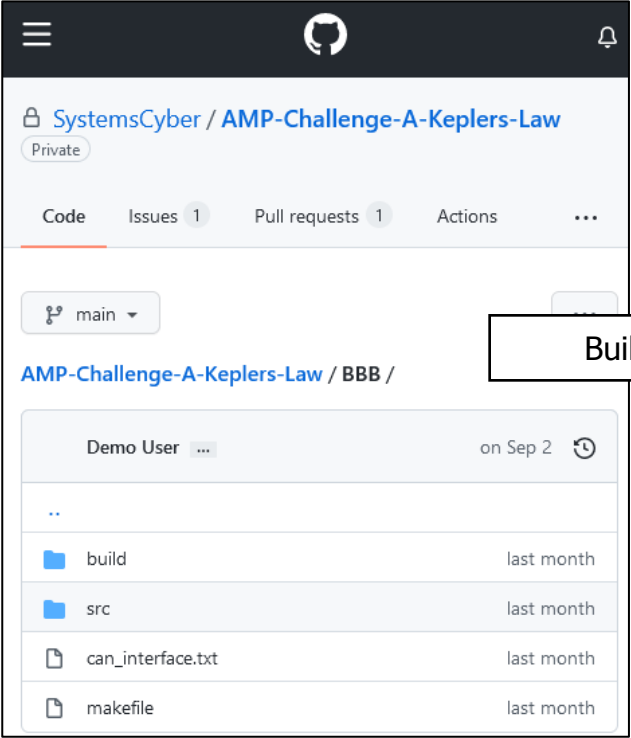


# Vision: Micropatches with certification evidence via automation



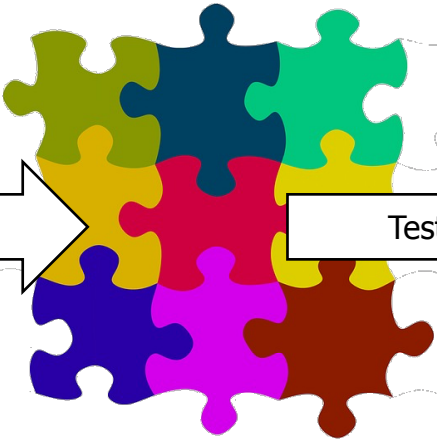
# Current industry practice

## Source code



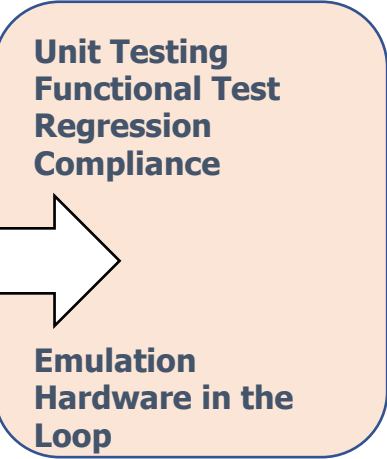
Source: Daily, May 2021

## Binary image

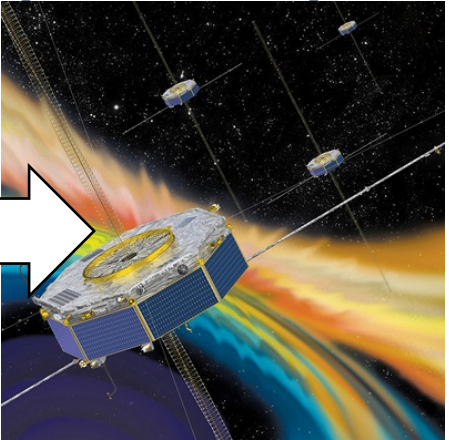


Source: Daily, May 2021

## Test & evaluation



## Operational system



Source: Azavea.com, August 2021

Build

Test

Pass?

Yes

Deploy

No

Fix

## Modify source code

```
#include <linux/can.h>
#include <linux/can/raw.h>
#include <string.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <unistd.h>
#include <sys/select.h>

// usage ./name [interface]
int main (int argc, char * argv[]) {

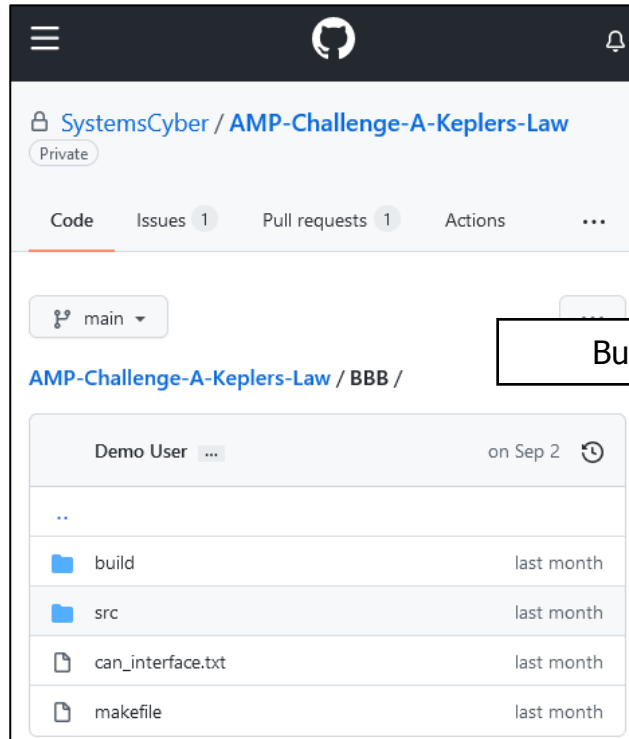
    struct ifreq ifr;
    struct sockaddr_can addr;
    struct can_frame cf;
    ssize_t nbytes;
    struct can_filter rfilter[2];

    int s = socket (PF_CAN, SOCK_RAW, CAN_RAW);

    strcpy(ifr.ifr_name, argv[1]);
```

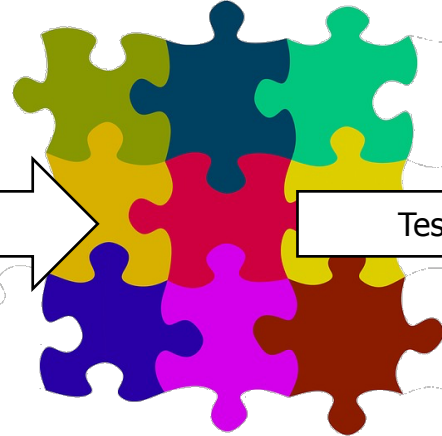
# Current industry practice

## Source code



Source: Daily, May 2021

## Binary image



Build

## Test & evaluation

Unit Testing  
Functional Test  
Regression  
Compliance

Test

Emulation  
Hardware in the  
Loop

Pass?

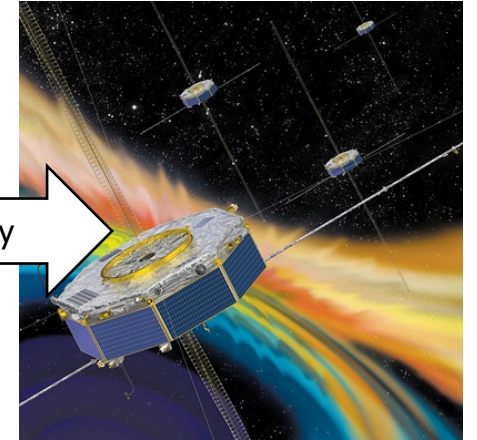
Yes

Deploy

No

Fix

## Operational system



Source: Azavea.com, August 2021

Source: Daily, May 2021

## Modify source code

```
#include <linux/can.h>
#include <linux/can/raw.h>
#include <string.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <unistd.h>
#include <sys/select.h>

// usage ./name [interface]
int main(int argc, char * argv[]) {

    struct ifreq ifr;
    struct sockaddr_can addr;
    struct can_frame cf;
    ssize_t nbytes;
    struct can_filter rfilter[2];

    int s = socket (PF_CAN, SOCK_RAW, CAN_RAW);

    strcpy(ifr.ifr_name, argv[1]);
```

## Challenges:

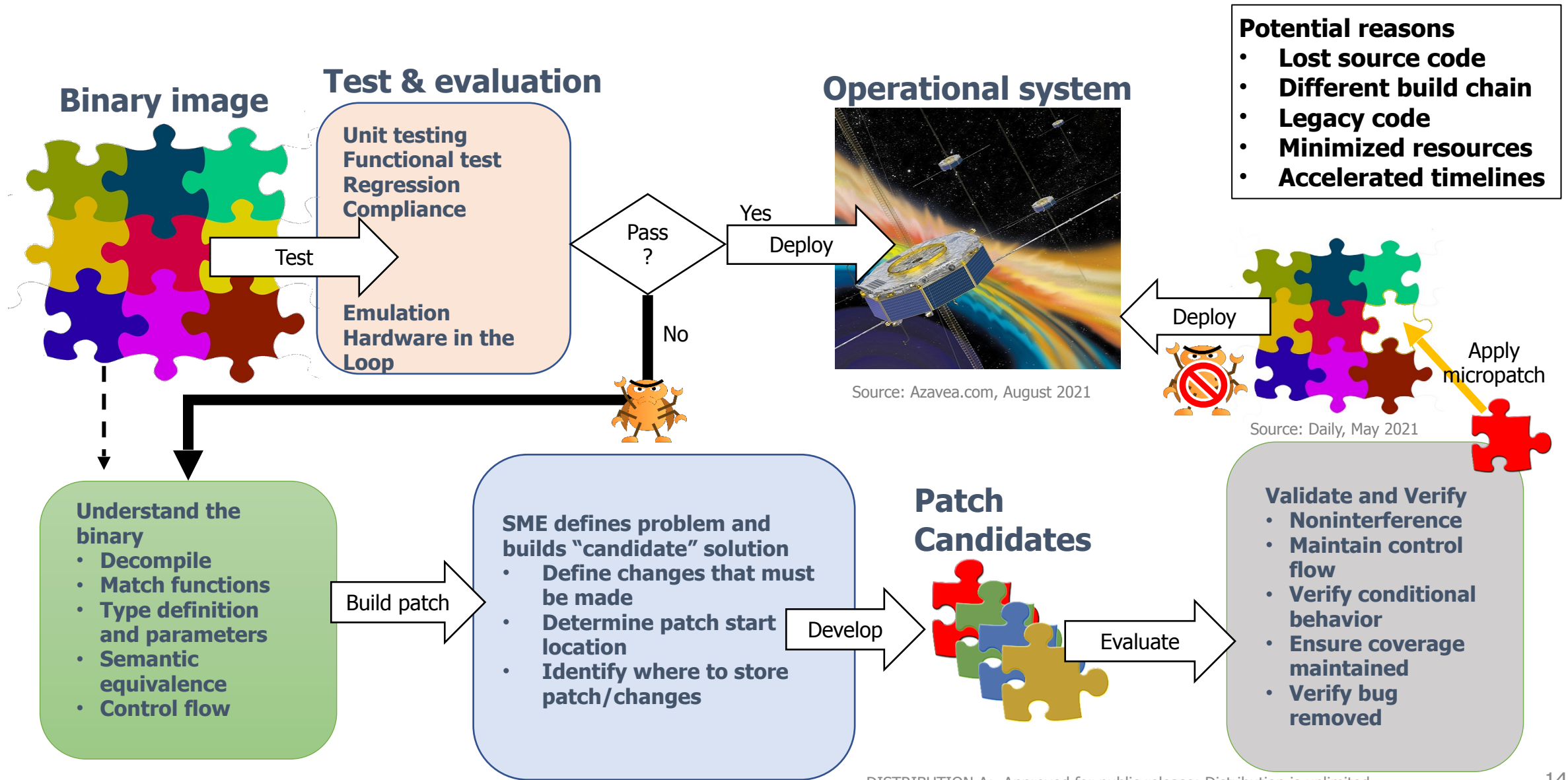
What if some source code isn't available?

- Boutique reverse engineering
- Binary rewriting

What if the build chain is not the same as before?

- Extensive re-testing, binary rewriting

# AMP technical approach





# Major technical challenges, ideas, and approaches

## TA1 Challenges:

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

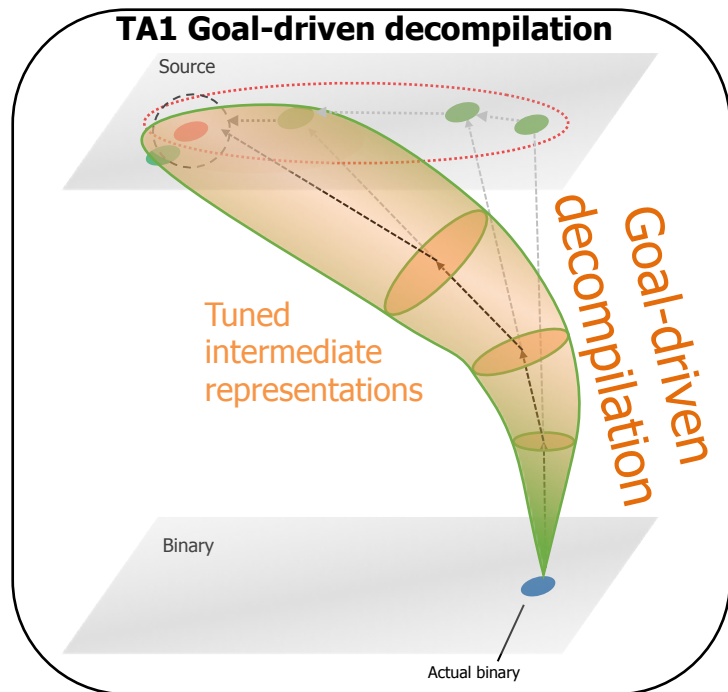
## TA2 Challenges:

- 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

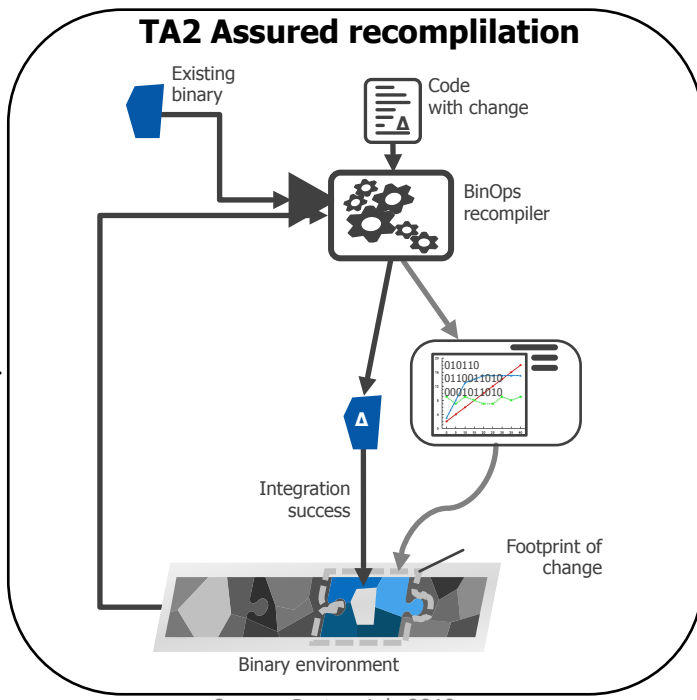
## TA1 Goal-driven decompilation



Source: Bratus, July 2019

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

## TA2 Assured recompilation



Source: Bratus, July 2019

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

## TA3 Evaluation



Source: Daily, July 2019

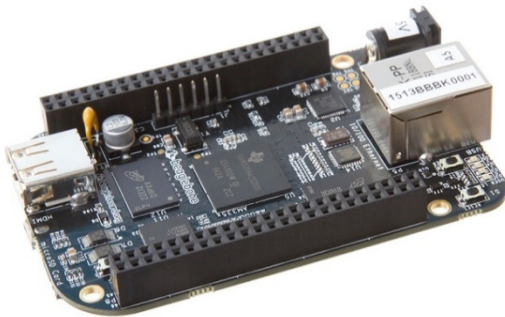
Approach: Representative heavy vehicle industry, which makes extensive use of embedded firmware for challenge problems

# TA3: Evaluation (heavy vehicle domain use case)

Provide tests of increasing difficulty culminating in networked system

## Phase 1: Commodity system

Development Boards

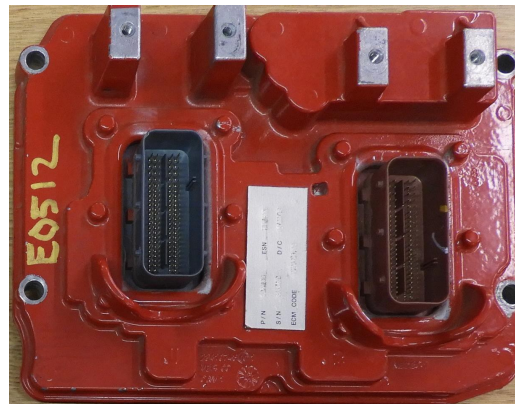


~1000-10000 lines of source code

[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)

## 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

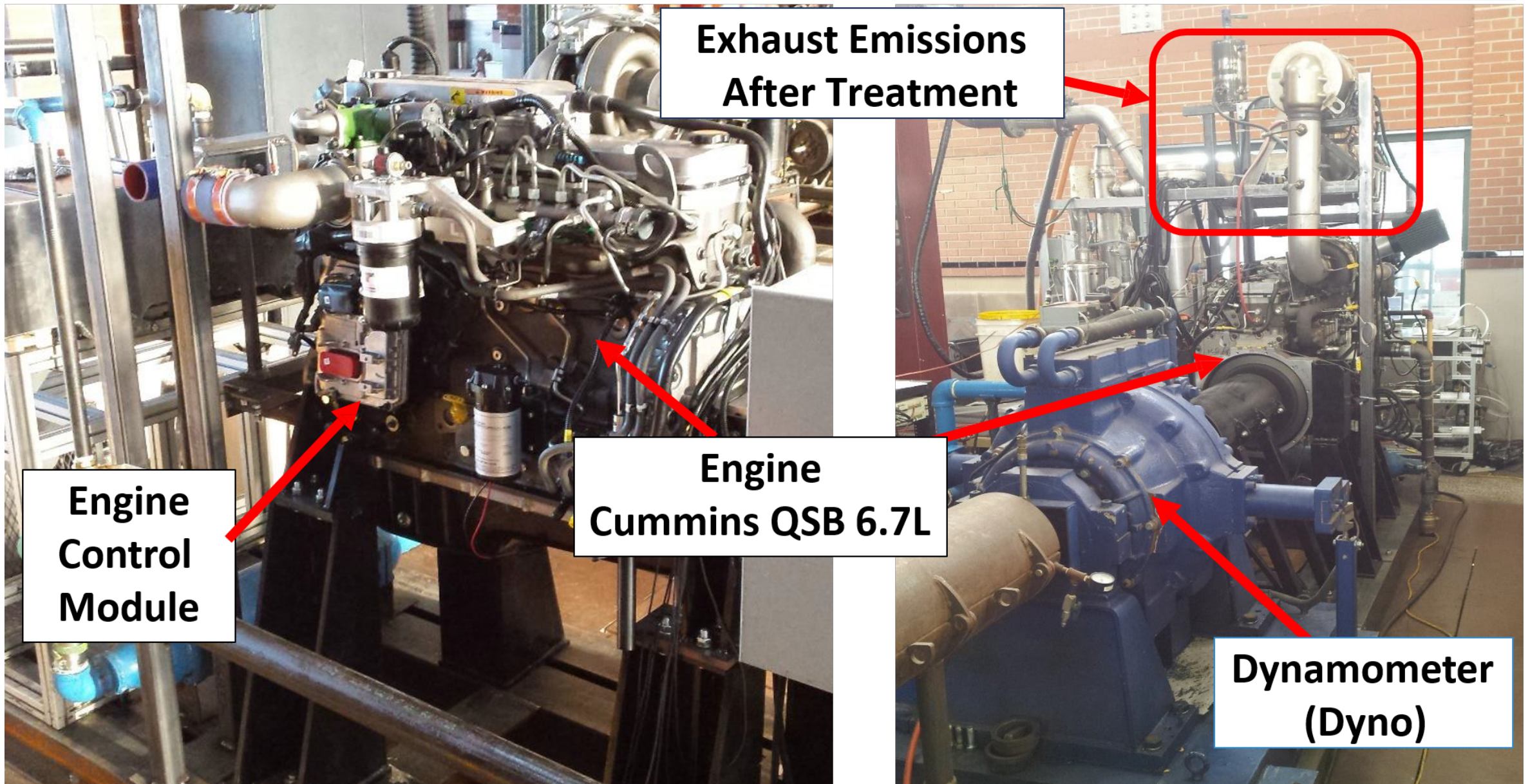


~5,000,000 lines of source code





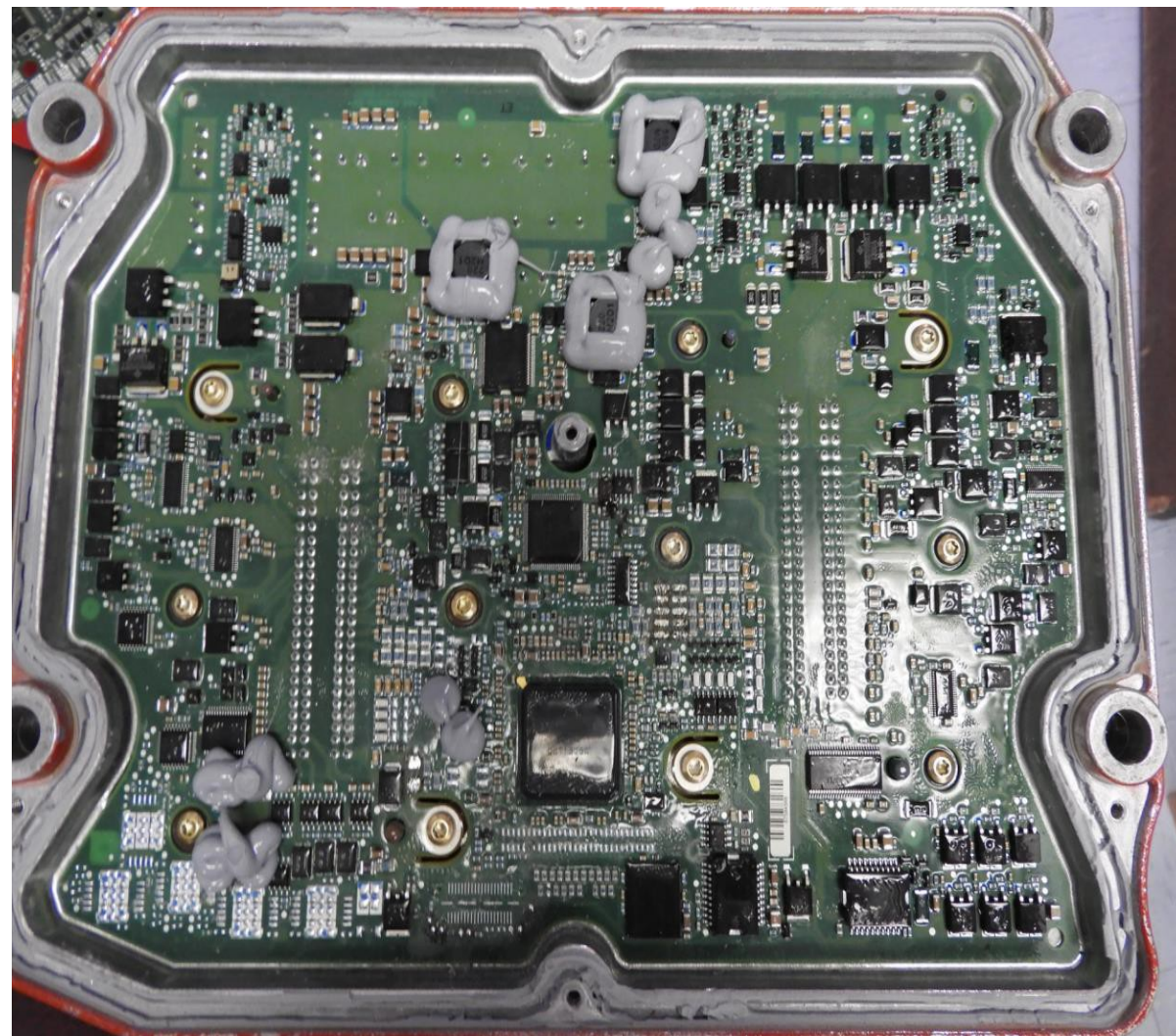
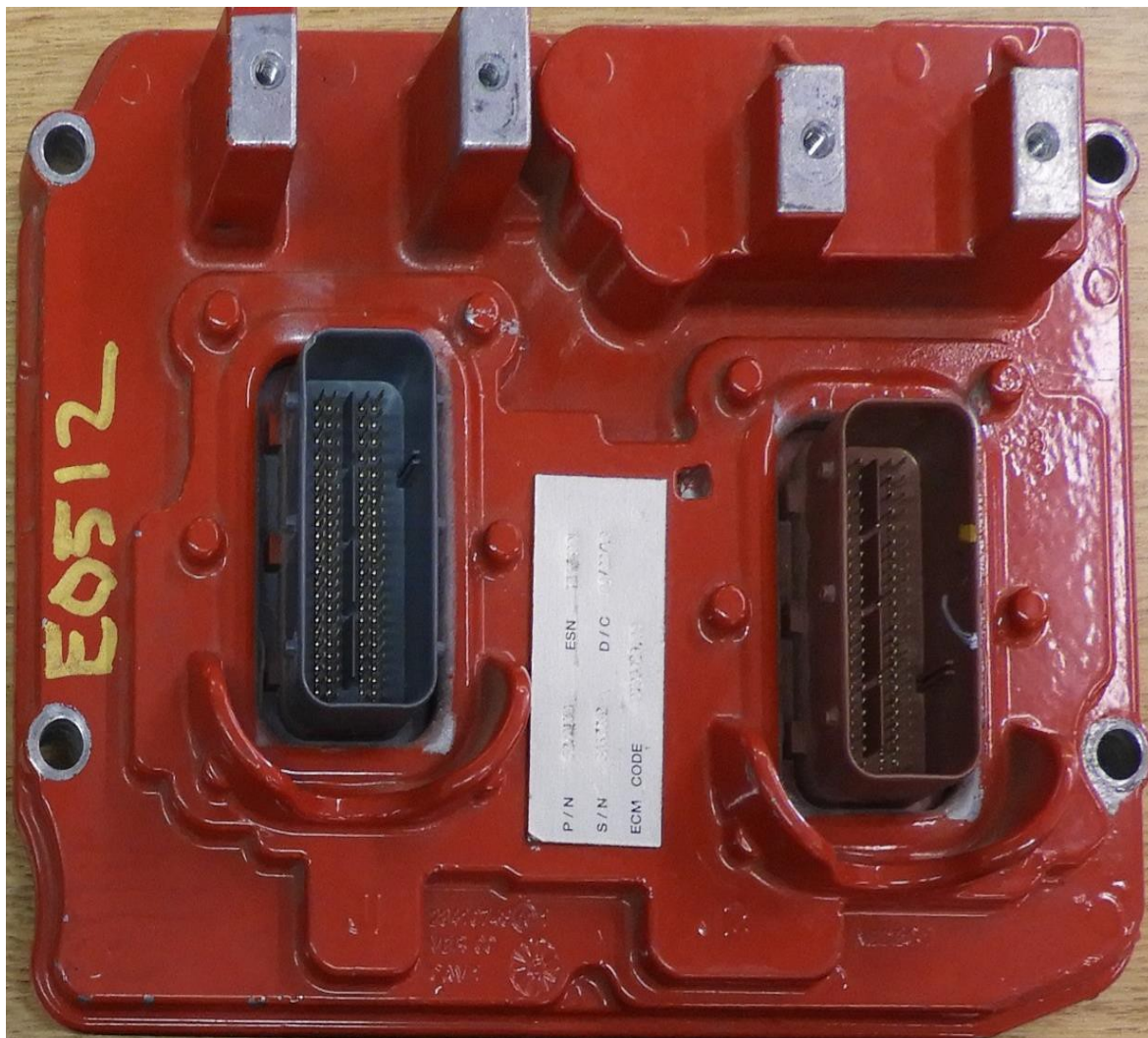
## AMP testbed at Colorado State University







## Engine Control Module (Cummins CM2350)





## A bit of history & a bit of vision

---

# Reverse Engineering ~ Math in RE





## 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).

# Reverse Engineering ~ IR tower/tree lifting?

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).

# Modular framework for binary research

## A new generation of tools for maintaining binaries

- **CodeCut**: <https://github.com/JHUAPL/CodeCut>
- **CodeHawk**: <https://github.com/static-analysis-engineering/codehawk>
- **VIBES**: <https://github.com/draperlaboratory/VIBES>
- **Remill, Anvil, Relic** LLVM lifters: <https://github.com/lifting-bits/remill>, <https://github.com/lifting-bits/anvill>, <https://github.com/lifting-bits/rellic>
- **PATE** binary patch verifier: <https://github.com/GaloisInc/pate>
- **MCTrace** code release: <https://github.com/GaloisInc/mctrace>
- Binary analysis and **rewriting** tools used by PATE and MCTrace:  
<https://github.com/GaloisInc/{macaw, reopt, what4, crucible, elf-edit, renovate}>, etc.

# Towers of Intermediate Representations

“IRs are useful. What’s an IR?”

- IRs are everywhere
  - LLVM passes ~ IRs, MLIR
  - Ghidra uses P-code
  - Angr uses VEX
  - Binary Ninja has 3 public IRs
- But what is an IR?
  - Trail of Bits: ..... *why only one?*

## Finding bugs in C code with Multi-Level IR and VAST

POST   JUNE 15, 2023   1 COMMENT

Intermediate languages (IRs) are what reverse engineers and vulnerability researchers use to see the forest for the trees. IRs are used to view programs at different abstraction layers, so that analysis can understand both low-level code aberrations and higher levels of flawed logic mistakes. The setback is that **bug-finding tools are often pigeonholed into choosing a specific IR**, because **bugs don’t uniformly exist across abstraction levels**.

We developed a new tool called **VAST** that solves this problem by providing a **“tower of IRs,”** allowing a program analysis to start at the best-fit representation for the analysis goal, then work upwards or downwards as needed. For instance, an analyst may want to do one of three things with a stack-based buffer overflow. (1) Identify it. (2) Classify it. (3) Remediate it.



# Bugs span the semantic gap, and so should analyses!

## Move up and down the tower of IRs as needed

Now comes choosing the right IR. Some bug properties are only apparent at certain abstraction levels. A buffer overflow is easily identified in LLVM IR, because stack buffers in LLVM IR are highly characteristic (i.e., created via the `alloca` instruction). This is the “best-fit” IR for identification.

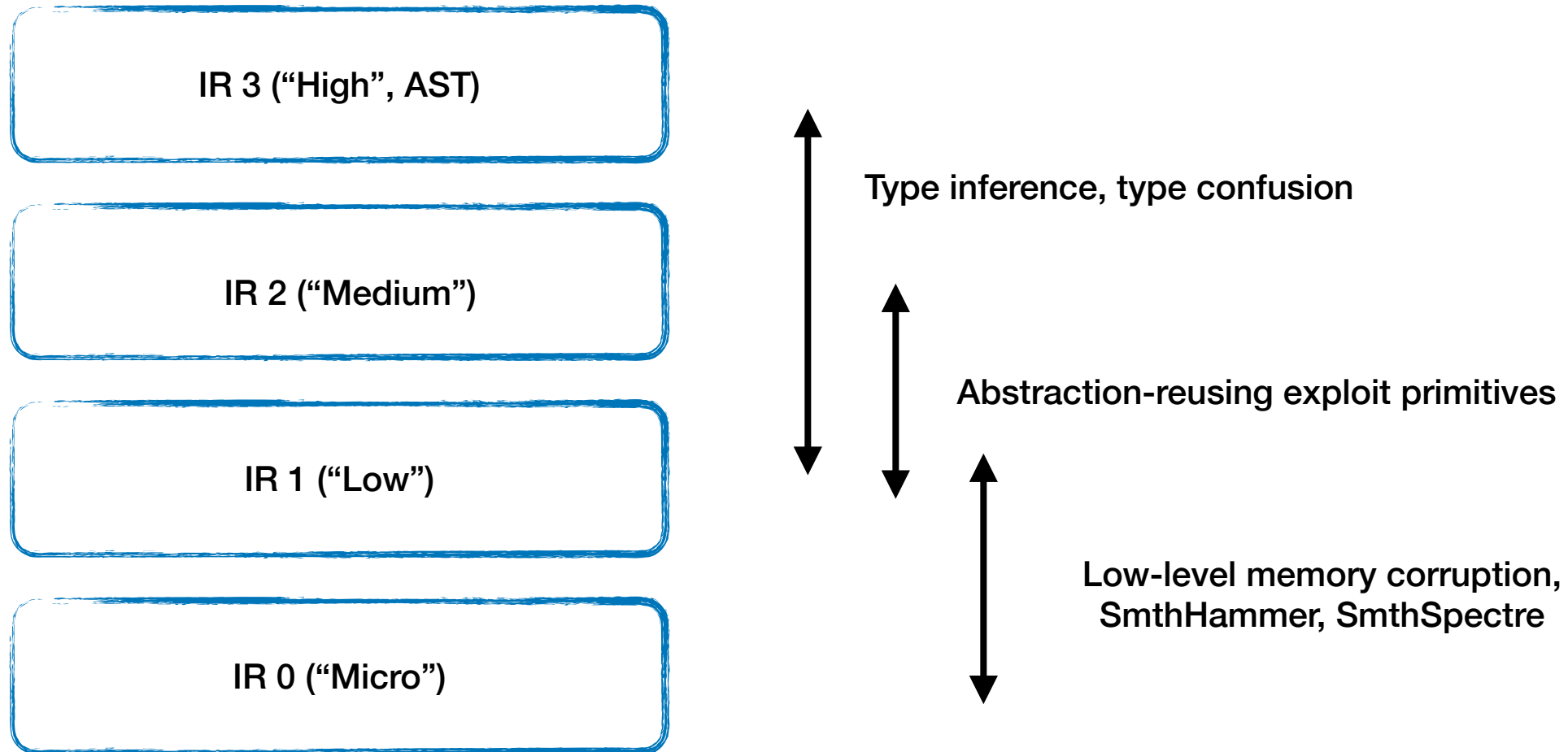
For *classification*, a buffer overflow can go from a common bug to a security threat if the buffer sits near sensitive data in program memory. This only becomes clear below the LLVM IR level, near or at the machine code level, where buffers are **fused together** with other sensitive information, forming a **“stack frame.”**

The last part of the story is *communication* and *remediation*. The reason why the buffer overflowed in the first place can be a **side-effect of a type conversion on a buffer index that was self-evident in the program’s abstract syntax tree (AST)**, the highest level IR. **Connecting these facts together used to be impossible, but VAST’s tower of IRs is changing this. Bugs span the semantic gap, and so should analyses.**

- Buffer overflows: LLVM IR
- Adjacency: below LLVM IR
- Root causes like out-of-type references: AST
- ToB solution: **VAST/Multiplier**
  - Get all the IRs (as dialects of MLIR)
  - “Move up or down as needed”

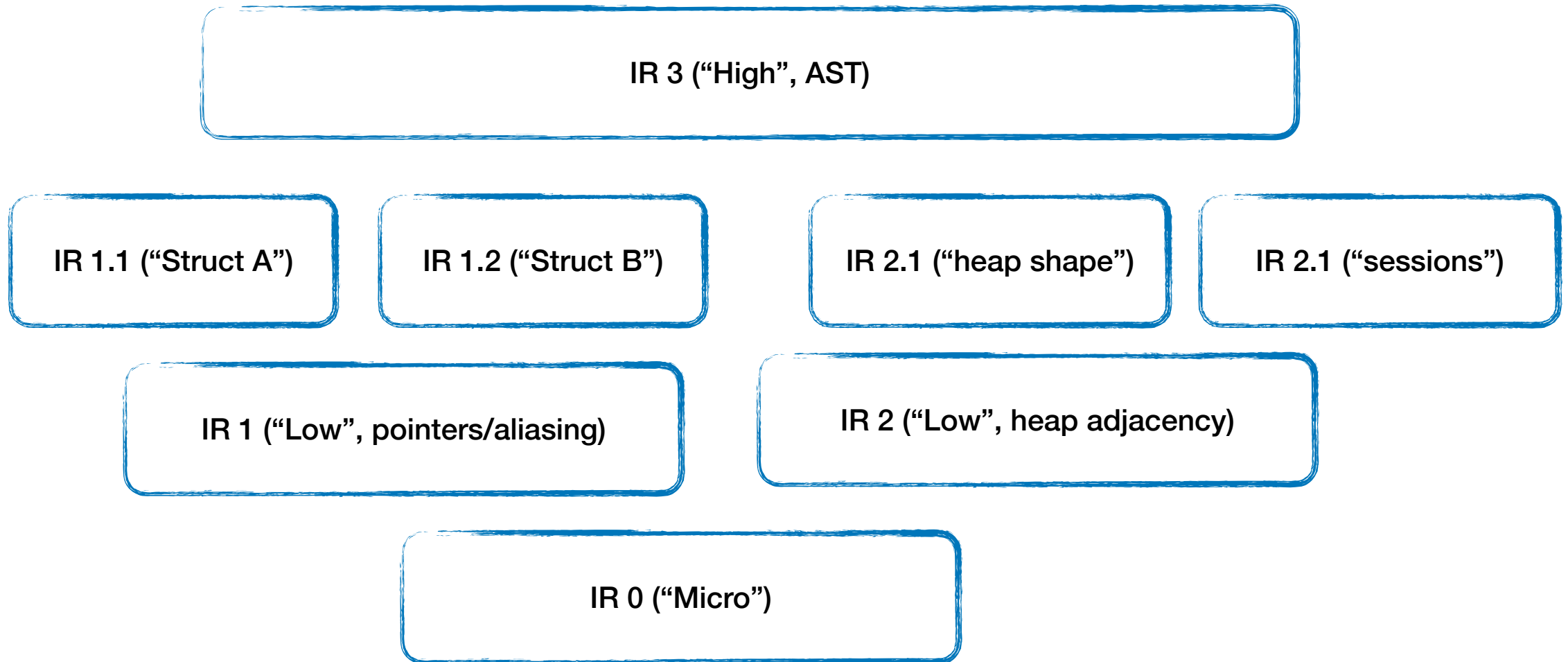
# A tower of IRs

A sequence of compatible, interoperating IRs



# A tree of IRs? A lattice of IRs?

“A sufficiently lifted IR is indistinguishable from a DSL”

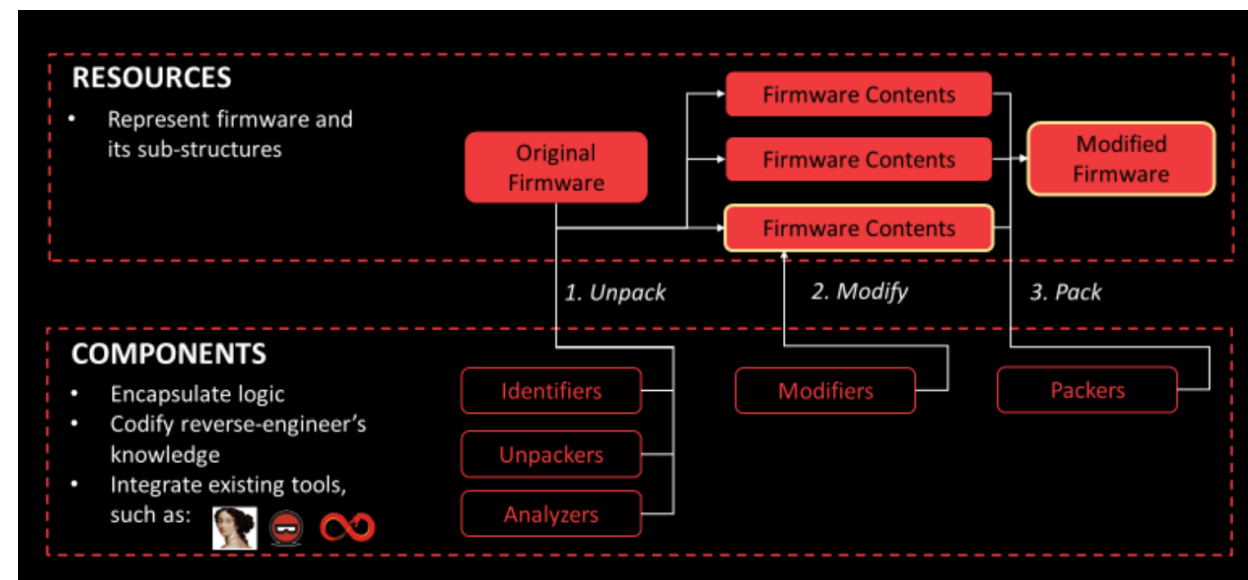
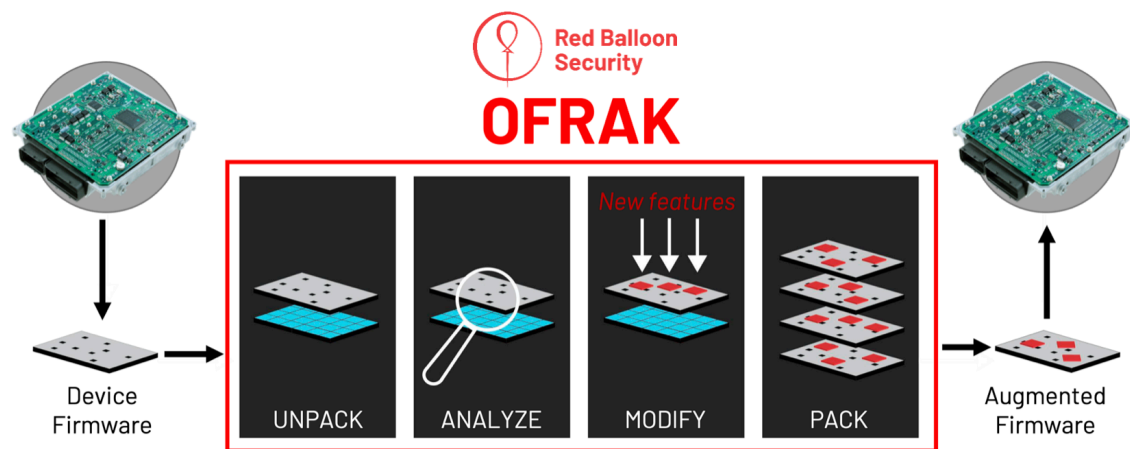


## AMP tools & prototypes

---

# Modular framework for decompilation research

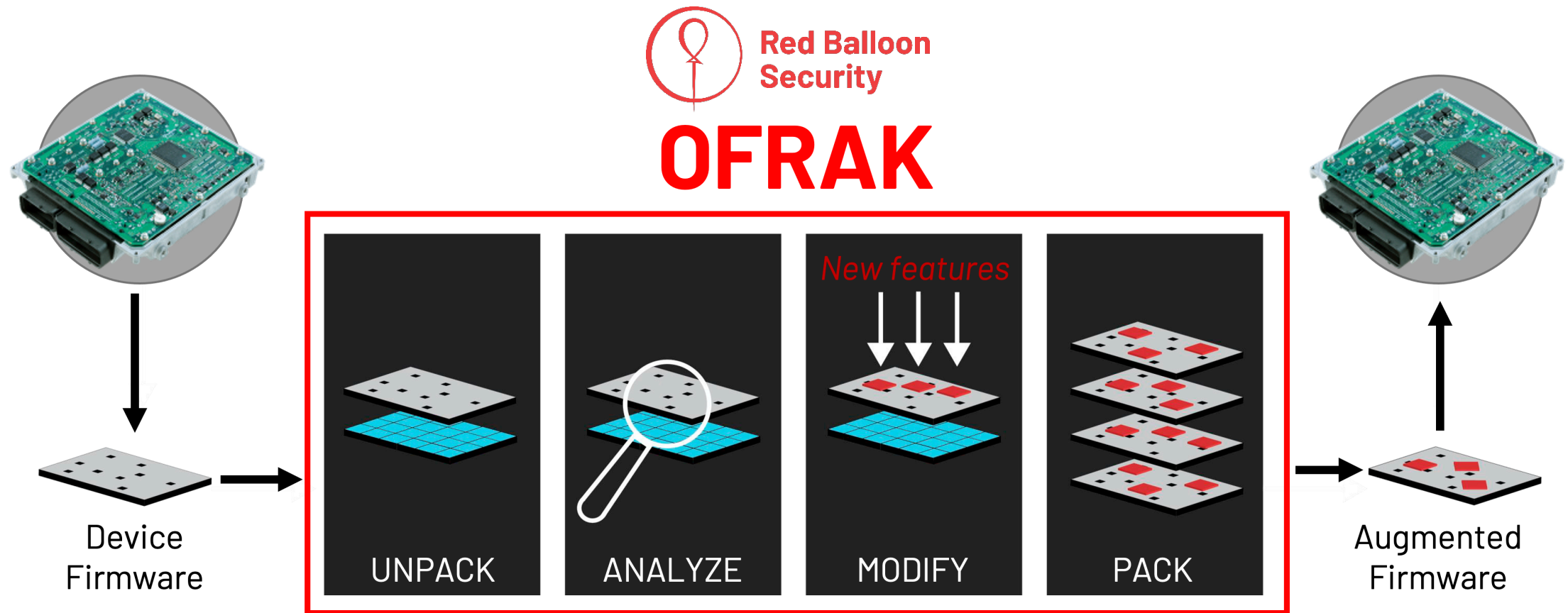
A new generation of tools for maintaining binaries



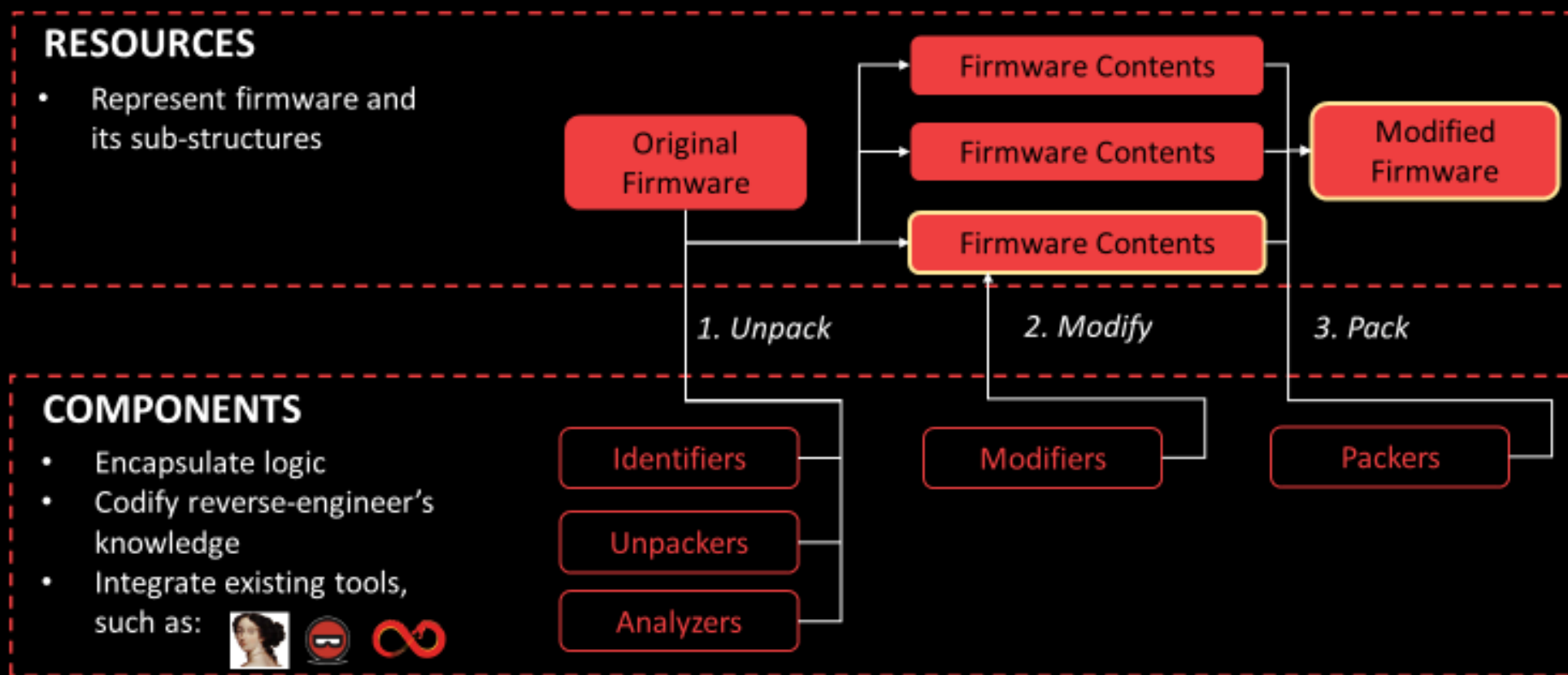
<https://github.com/redballoonsecurity/ofrak>



# Unpack, Analyze, Modify, Pack Workflow



# OFRAK cont.



## OFRAK cont.

## Resource Tree Pane

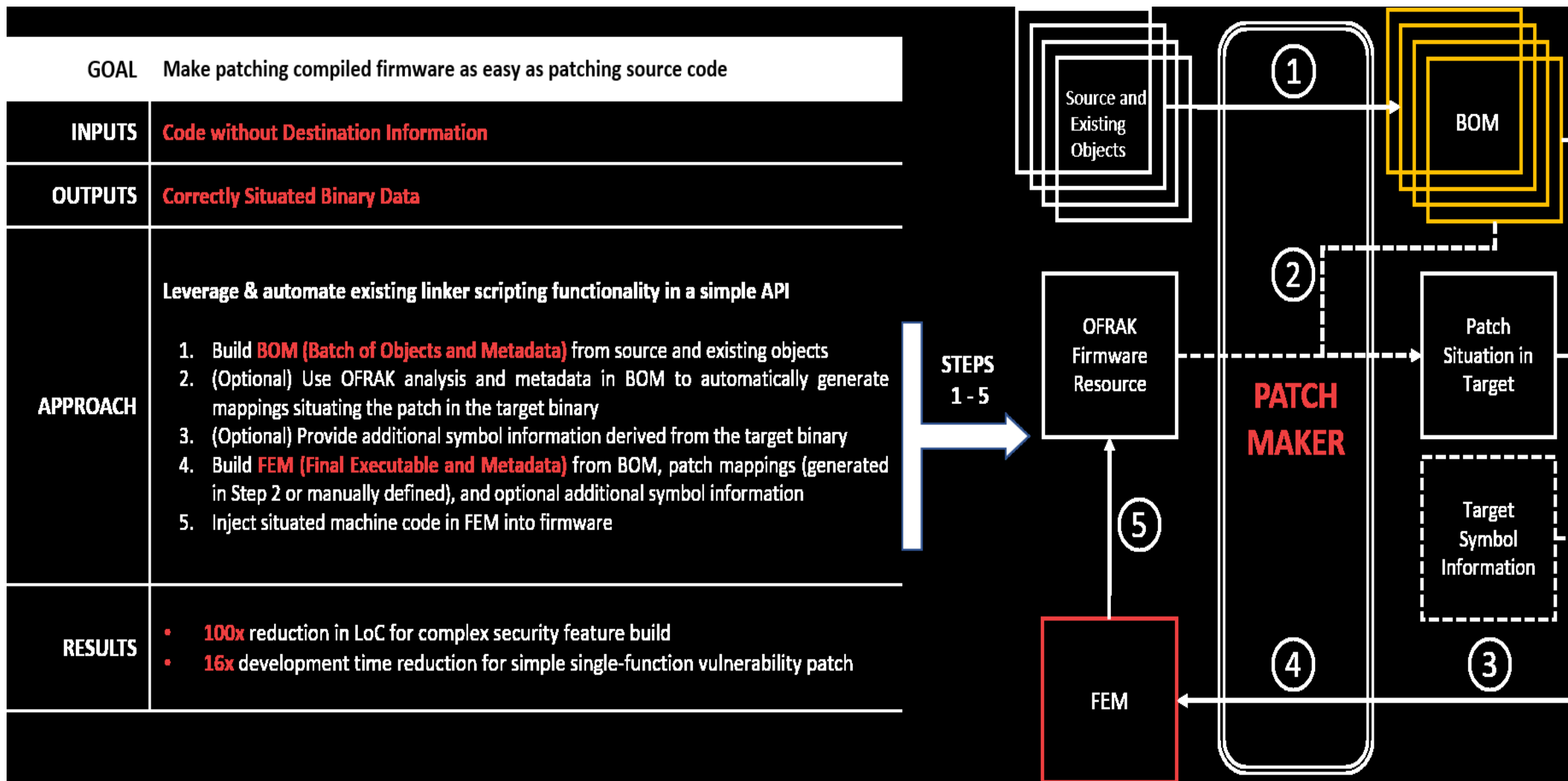
## Hex Pane

The screenshot displays the Binary Ninja interface with several key components highlighted by red dashed boxes and arrows:

- Action Menu:** Located on the left, it contains buttons for Unpack, Unpack Recursively, Download, New, Identify, Analyze, Modify, Pack, and Pack Recursively.
- File Tree:** A hierarchical view on the left showing the file structure. The selected item is "BinaryNinjaAnalysisResource, File: hello\_elf, Elf".
- Tags and Attributes:** A panel on the left showing the tags (BinaryNinjaAnalysisResource, GenericBinary, FilesystemEntry, File, Program, Elf) and attributes (FilesystemEntryAutoAttributes) for the selected file.
- Disassembly View:** The main area on the right showing the assembly code. The selected instruction is "00000000: 7f 45 4c 46 01 01 00 00 00 00 00 00 00 00 00 00 .ELF.....".

## Resource Details Pane

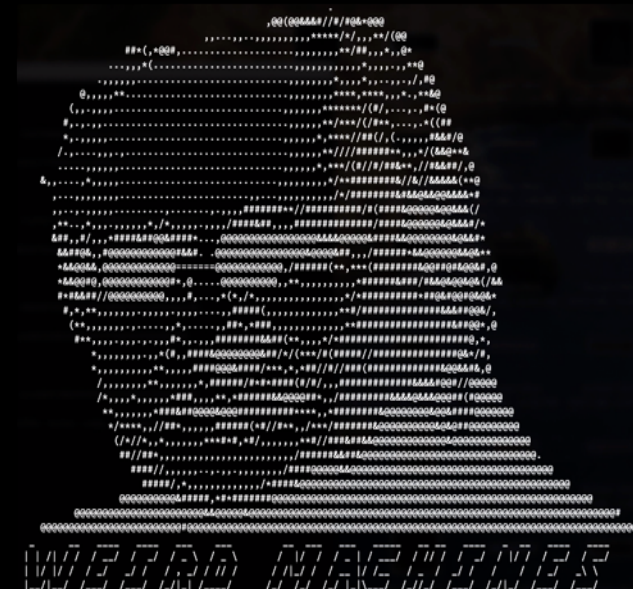
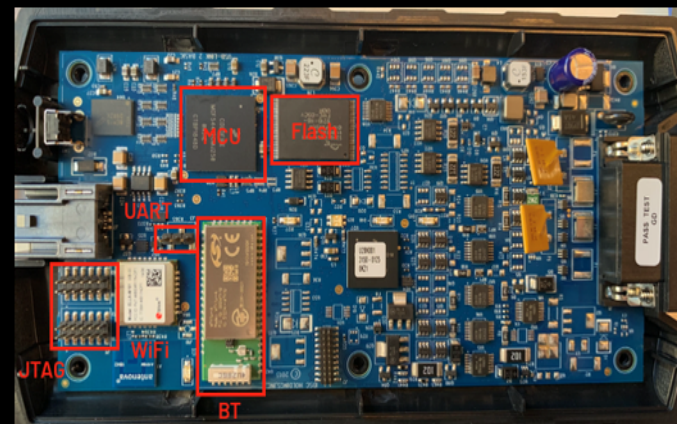
# OFRAK Patch Maker



# Diagnostics adapter autotomy

[autotomy\_modifier.py: 137] Autotomy applied to 574 functions and 0 data ranges from 24 entry points (0x23fbc bytes code, 0x0 data)

PROBLEM	Ground vehicle diagnostics adapter used by DoD customer(s) comes packaged with Bluetooth and WiFi connectivity
CHALLENGE	Can we <b>remove unwanted features</b> from firmware with <b>no source code access</b> ?
APPROACH	Use <b>OFRAK</b> , a modular firmware reverse engineering and analysis framework built by Red Balloon Security (RBS) under DARPA AMP, to apply RBS's proprietary Autotomy algorithm to remove unused and unwanted features.
RESULTS	<ul style="list-style-type: none"> <li>• <b>Permanently disabled Bluetooth and WiFi</b> in firmware by removing all associated code</li> <li>• <b>Removed 144KB</b> of code / added free space</li> <li>• Modifying device firmware was extremely easy – no manufacturer signature to bypass</li> </ul>



*Top Right: Hardware teardown results of the ground vehicle diagnostics adapter*

*Bottom Right: Example outcome of a payload taking advantage of free space with ASCII animation playing instead of normal ping function*



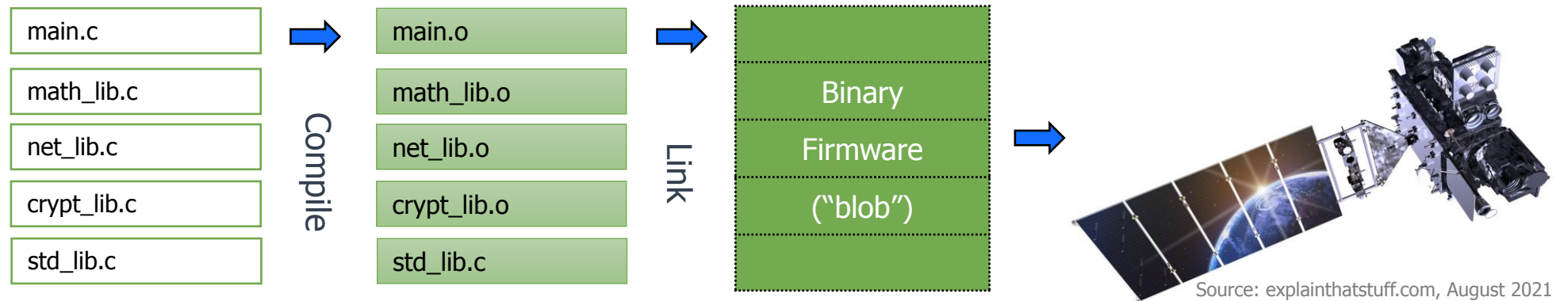
## De-linking a binary back into modules

---

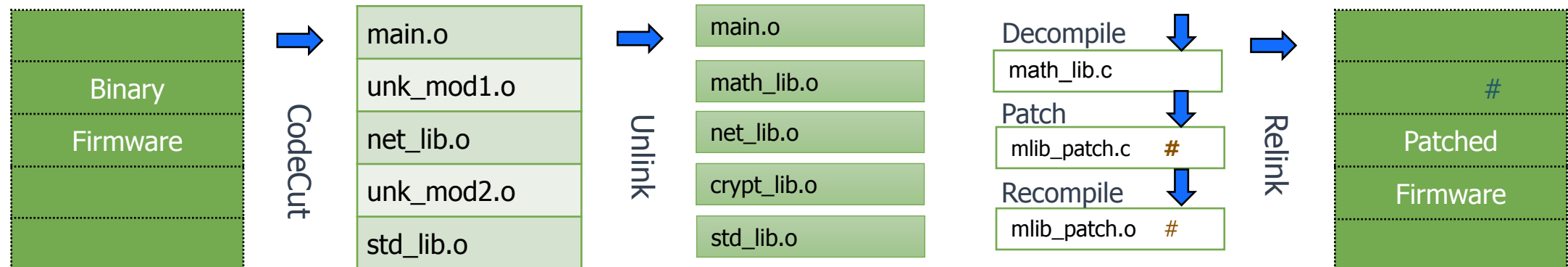
# John Hopkins University/APL **CodeCut**

- Current build process for embedded systems & Cyber Physical System's (CPS) firmware is one-way
- Reversing to patch or otherwise modify a binary is manual, very labor-intensive, and disjoint from the software development ecosystem
- AMP envisions "unlinking" and "relinking" along with improved decompilation to make binary patching faster and approachable to non-experts

## Traditional build process

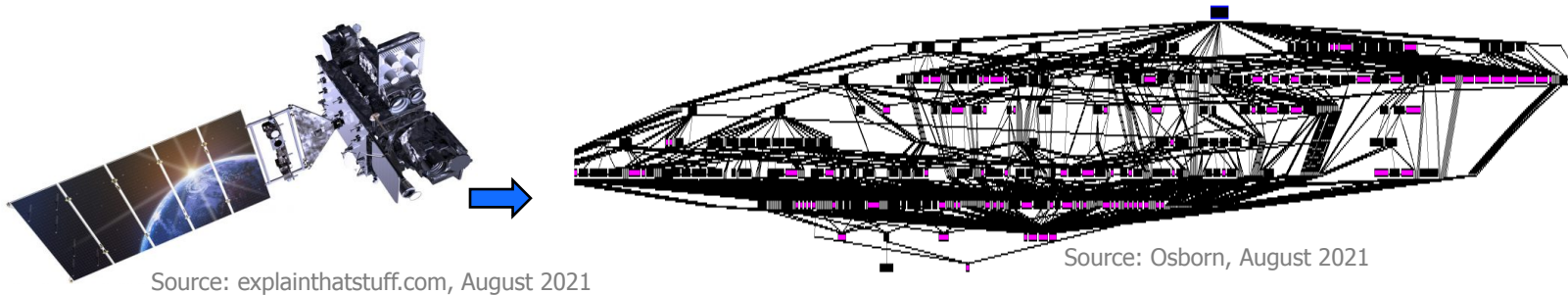


## AMP envisioned process



# John Hopkins University/APL CodeCut – Decompilation for select object files

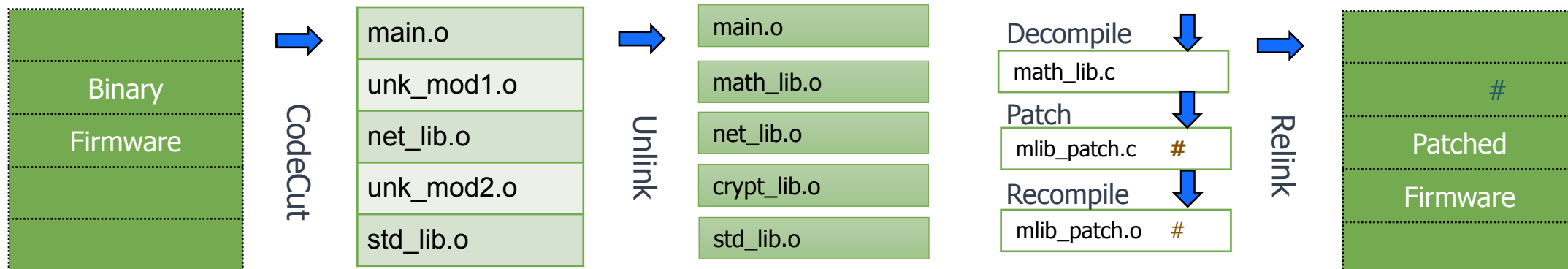
- Background: Reverse engineers currently operate on a function level because there is no automated way to recover module (object file) boundaries within a fully-linked binary -- Decompilation has to happen at a function or full-program level



## AMP Tools Status:

- Implemented **deep learning** model that improves accuracy over statistical approaches
- Full **Ghidra** implementation
- Working towards module-level decompilation

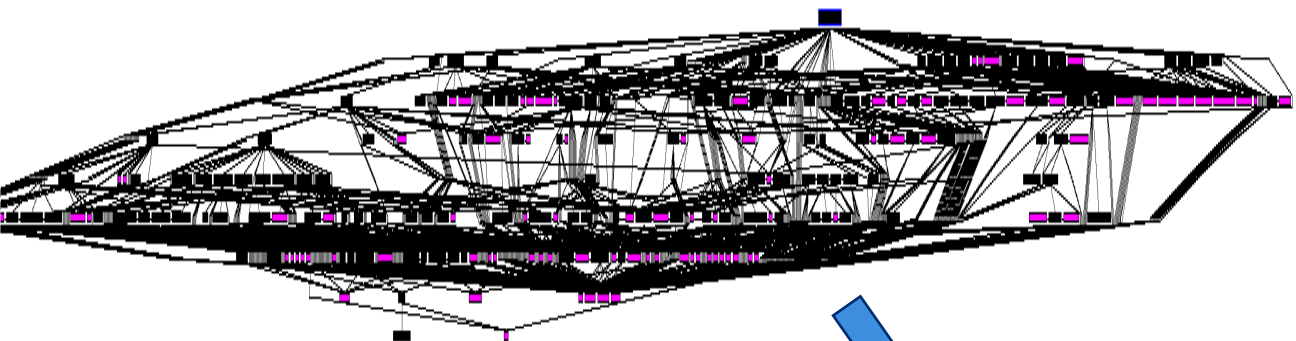
- Problem Statement: Given only call graph information for a large binary, recover the **boundaries of the original object files**



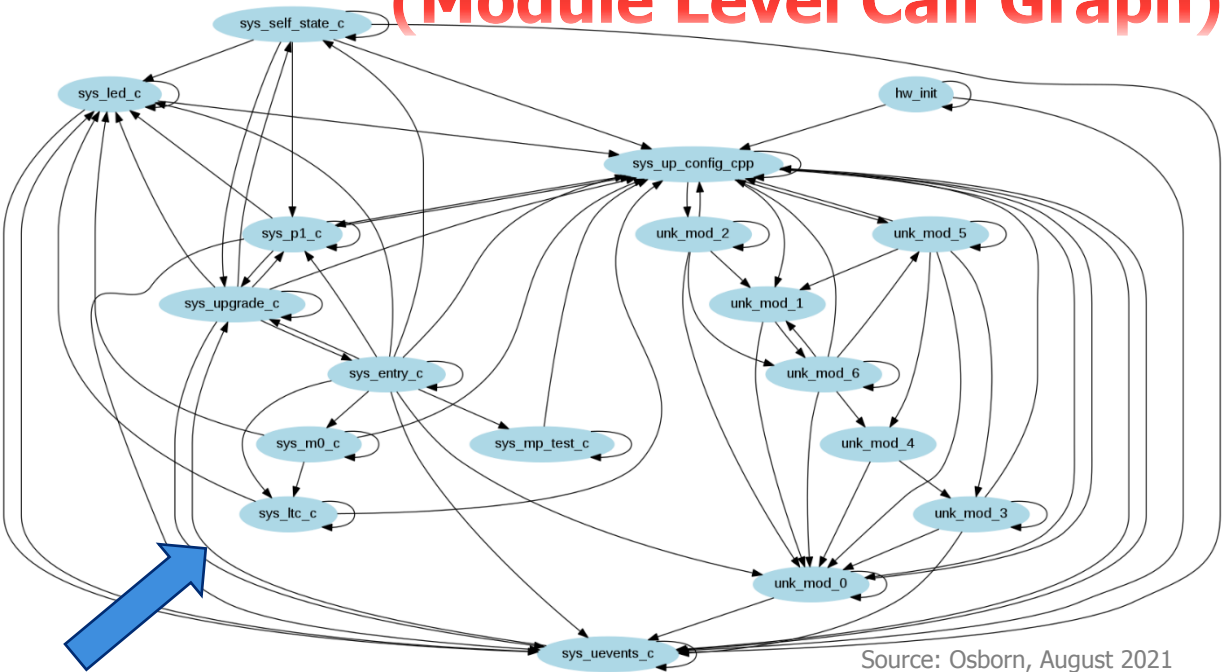
Source: Osborn, August 2021

# What is the impact?

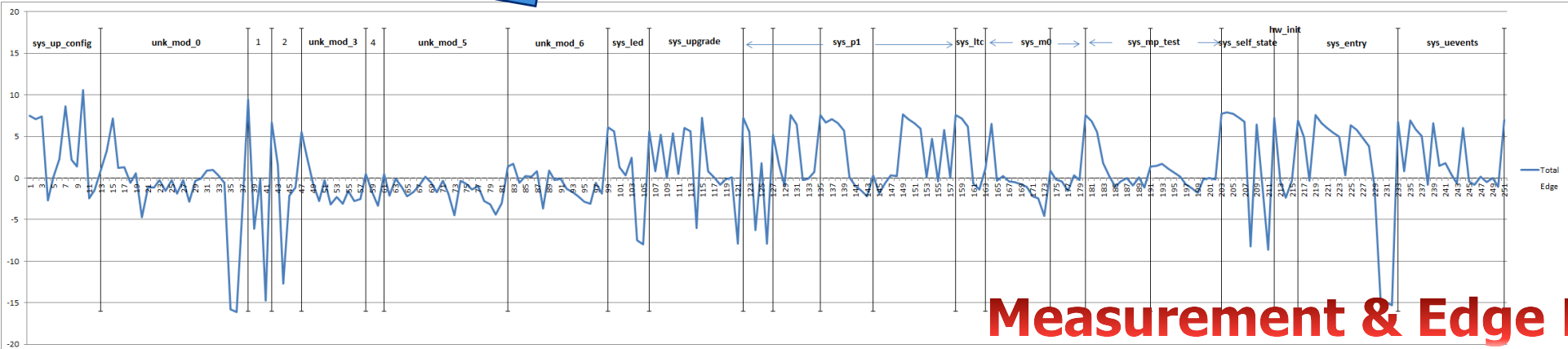
## Software Architecture (Module Level Call Graph)



Source: Osborn, August 2021



Source: Osborn, August 2021



Source: Osborn, August 2021

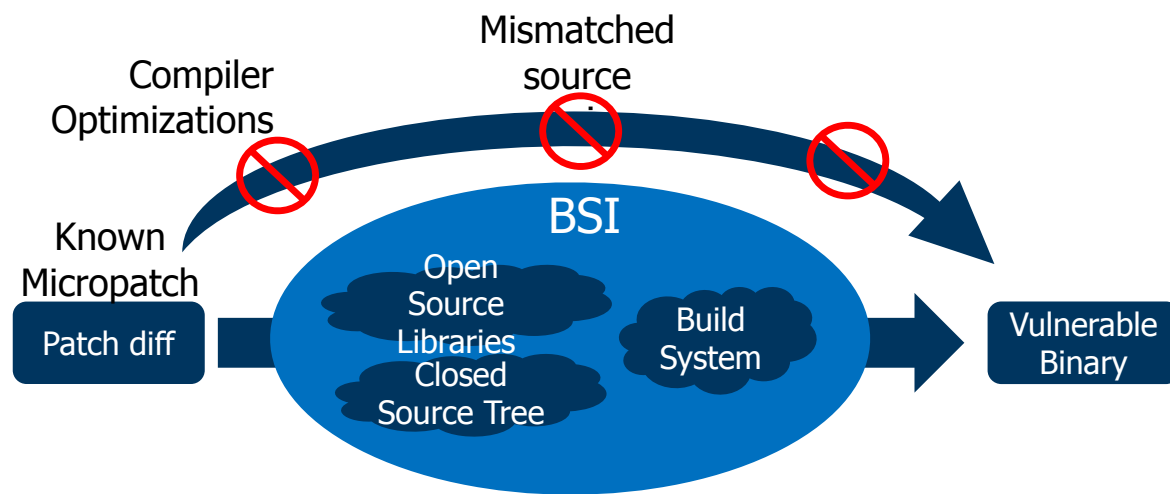


## Aligning available source code with the binary

---

# Binary Structure Inference (BSI) concept of operations

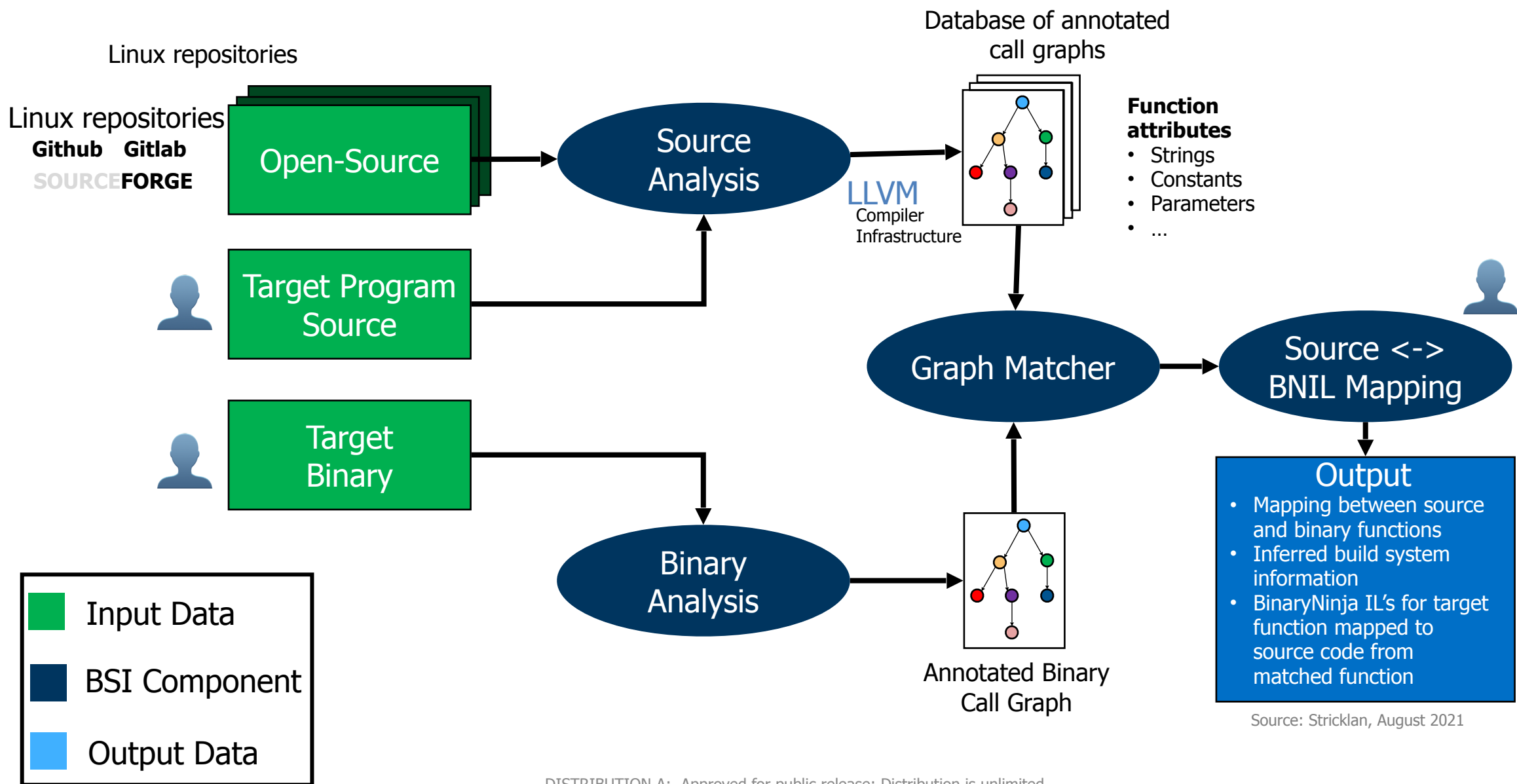
- BSI assists in micropatch localization by **aligning source code** to a **target binary** using a probabilistic graph matching algorithm
  - Unknown information about the original build system and program version can complicate the process of aligning a deployed binary back to its original source code
  - BSI handles this problem by searching over a **range of build configurations** and program versions
  - The core graph matching algorithm assigns a **source function** to **each binary function** in a way that preserves individual function attributes as well as overall program structure
- BSI performs detailed analysis of where source code patches effect a target binary
  - BSI enables users to **quickly import source information** into their binary inspection tools
  - The user can create mappings between the source code and a lifted Intermediate Representation (IR) to **enable** more **granular analysis**
  - Using the aligned source code and IR mappings, BSI can perform detailed analysis on **where a given source code patch will affect the binary**



BSI will help reason over partial unknowns that affect patch location



# Binary Structure Inference (BSI) system architecture



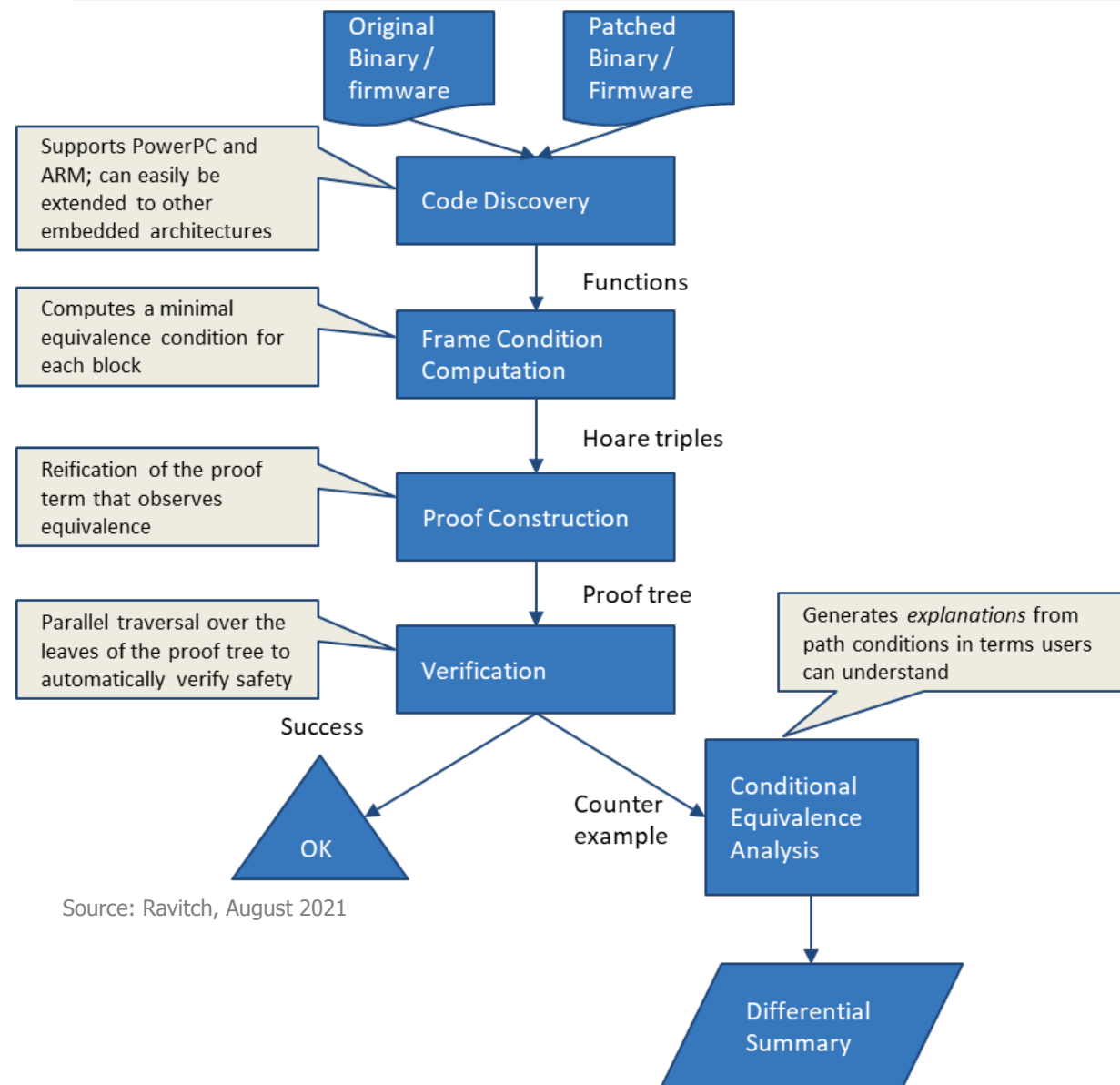
Source: Stricklan, August 2021

## Relational analysis: Explaining behavior differences after patch

---



# Patches assured up to trace equivalence (Galois)



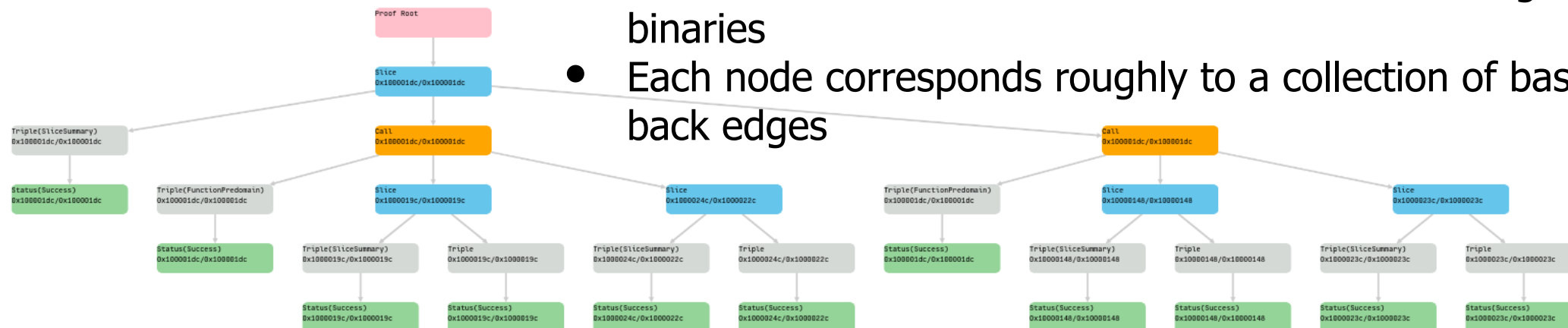
The Galois PATE tool is an automated **relational verifier** that **explains the differences** between two binaries or firmware images (e.g., an original and patched binary)

It reduces the time required to build high-assurance binary patches for embedded systems and avionics by:

- **Automatically applying** static program verification techniques
  - Brings the power of formal reasoning to the traditionally risky binary patching process, improving confidence in patches that must work on the first try
- **Extracting** specifications from the original binary
  - Removes the need for users to manually write detailed specifications, while being able to incorporate specifications if desired
  - Does not require hardware models, thus enabling it to handle complex/custom embedded hardware interfaces
- **Explaining** the impact of changes in terms of differences in observable behavior
  - Makes explanations of patch effects understandable to domain engineers without a background in verification or reverse engineering; includes an interactive proof visualization
- **Classifying** changes as benign when they only improve the program (e.g., removing known-bad behaviors)
  - Improves confidence in bug fixes for high-value systems and reduces the analysis burden for safe patches

# Interactive equivalence proofs

- Proof is reified as a data structure that can be visualized
- Leaves are proof goals, where failures (red and yellow) represent observable behavioral differences between the original and patched binaries
- Each node corresponds roughly to a collection of basic blocks without back edges



Triple  
0x10df4/0x10df4

Status(Inequivalent)  
0x10df4/0x10df4

A basic block that always exhibits different behavior in the patched binary; note that this may be intended (depending on the nature of the patch)

Triple  
0x10cf4/0x10cf4

Status(Conditional)  
0x10cf4/0x10cf4

A basic block that sometimes exhibits different behavior in the patched binary; this is accompanied by a differential summary that explains the conditions under which the behavior differs

The summary is a first-order logical formula over function inputs

# Verification strategy

Automatically assembles individual frames into a large *compositional* equivalence proof

Note that PATE attempts to prove equivalence, but the proof is not expected to succeed because the patch *should* change the program's behavior

The goal of the proof is to generate an explanation

Frame Condition  
Computation

Hoare triples

Proof Construction

Proof tree

Verification

Counter  
example

Conditional  
Equivalence  
Analysis

Frame conditions are all of the program state that must be equivalent after an (original, patched) block pair execute in order for their effects to be equivalent:

- Registers
- Stack memory
- Other memory

The stack is treated specially to improve proof compositionality (i.e., it is expected that stack frames are mostly isolated)

The frame computation is automated and based on inter-procedural context-sensitive demand analysis; demanded values must be equivalent between the original and patched programs (at observable events like system calls and memory writes)

Traverses the leaves of the proof tree (proof obligations) in parallel, enabled by compositionality

Proof obligations are discharged automatically using SMT solvers

Generates an explanation of the conditions under which the patched program exhibits different behavior

Attempts to classify changes as benign where possible (e.g., if a patch removes only bad behaviors from the program, it is benign)

Bad behaviors are "free properties" (e.g., memory errors) or user-specified states that should not be reachable

Source: Ravitch, August 2021

## Patching a binary at a higher level of abstraction

---



**Problem:** Directly patching legacy, stripped binaries is insanely difficult, expensive and error-prone

```
0xcc STR    R1,[R7, #0x0]
0xce LDR    R3,[R7, #0x0]
0xd0 LDRB   R3,R3,[R3, #0x5]
0xd2 STRB   R3,[R7, #0xf]
0xd4 LDR    R3,[R7, #0x4]
0xd6 ADDS   R3,R3,0x3
0xd8 LDRB   R3,R3,[R3, #0x0]
0xda UXTH   R3,R3
0xdc LSL    R3,R3,0x8
0xde UXTH   R2,R3
0xe0 LDR    R3,[R7, #0x4]
0xe2 ADDS   R3,R3,0x2
0xe4 LDRB   R3,R3,[R3, #0x0]
```

## Solution:

1. We lift binary to familiar C-like code
2. You directly patch the C-like code!
3. We automatically translate to a **minimally-invasive** patch on the binary

```
if (brake_switch != 0) {
    if ((speed_value != 0 &&
        {speed_value < 0}) {
        if (bumper[4] == 0) {
            bumper[6] = 1;
        }
    }
} else {
    bumper[6] = 0;
    bumper[4] = 0;
}
```

Patched  
EXE

# How to assure patch fixes the problem?

**Problem:** How do you assure the patch correctly fixes the problem, does not break the desired behavior and has no unintended consequences?

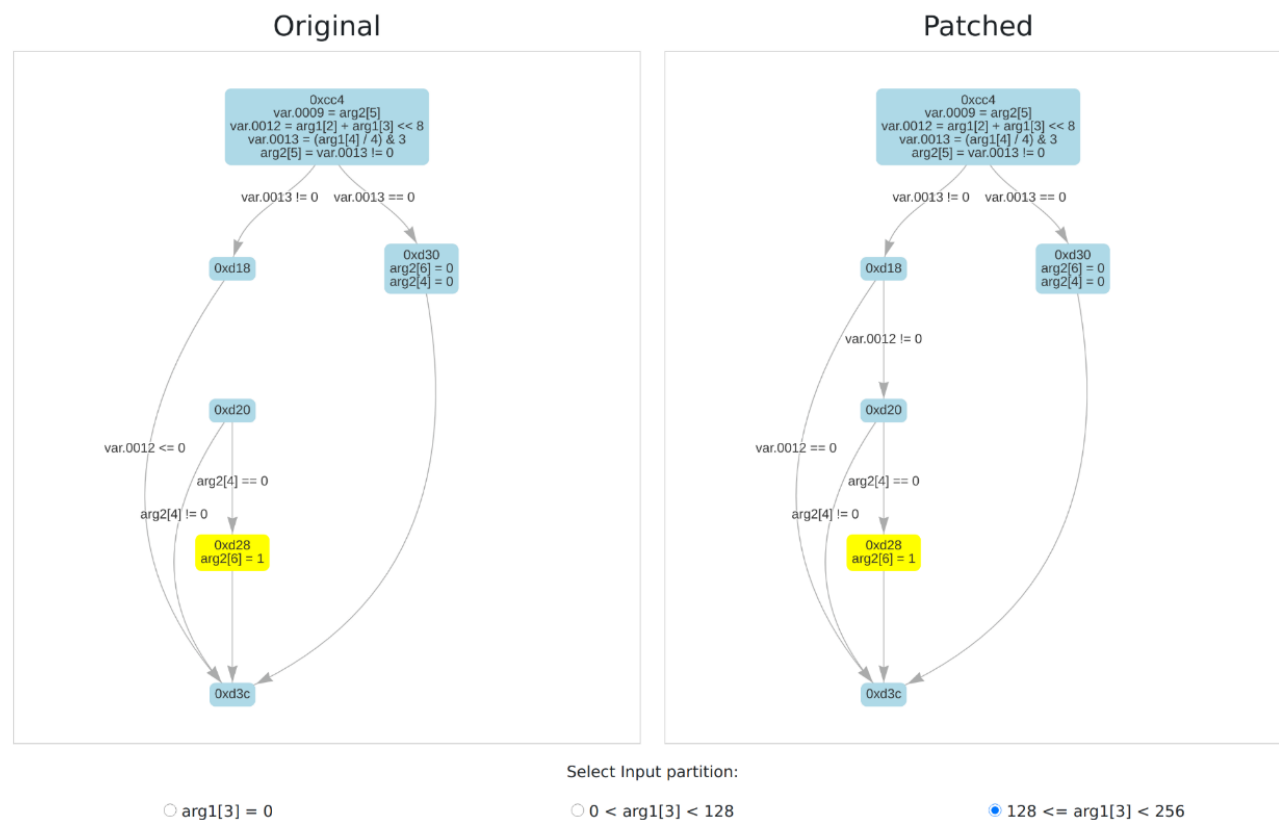
## AMP MRAM Solution:

1. We analyze original versus patched binary
2. We precisely track the differences
3. We provide intuitive details of how program actions differ with varying input
4. You decide if these details represent a correct patch ... much easier than reasoning about code!

## AMP Challenge 3 Example

Global Analysis:

- 20 functions unchanged
- 1 function changed:



# Multifocal Relational Analysis for Assured Micro-patching (MRAM)

---

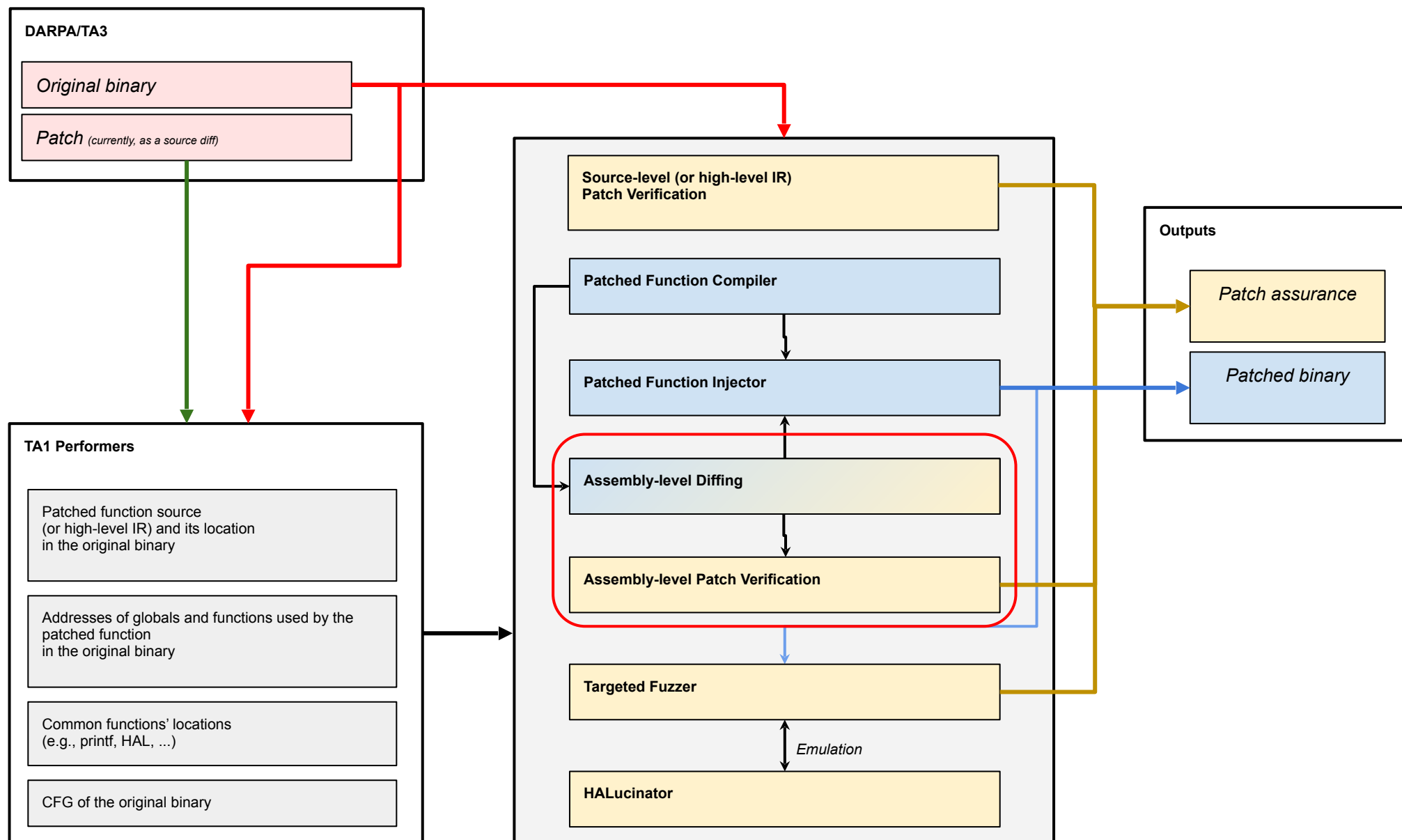
Advanced binary abstract interpretation engine enables our techniques

- Recover source-level constructs from binary
- Invariants on the values of program variables

Progress on AMP Challenges:

- Automatically generated patches from manual changes on lifted code
  - Minimal binary disruption: Average patch changes only 4 instructions
- Automated relational analyses denotes how functions have changed
  - Structural invariant, control-flow invariant, etc.
- Automated relational analysis demonstrates how functions have changed based on function input and program state / actions

# PatcherX (Purdue U. / EPFL)





- DisPatch
  - Demonstration: Patching ArduPilot Drone Firmware
    - Firmware dumping
    - DisPatch workflow
    - Validation: Roll rate reference enforcement
    - Validation: Roll P parameter enforcement

# Tool summary

---

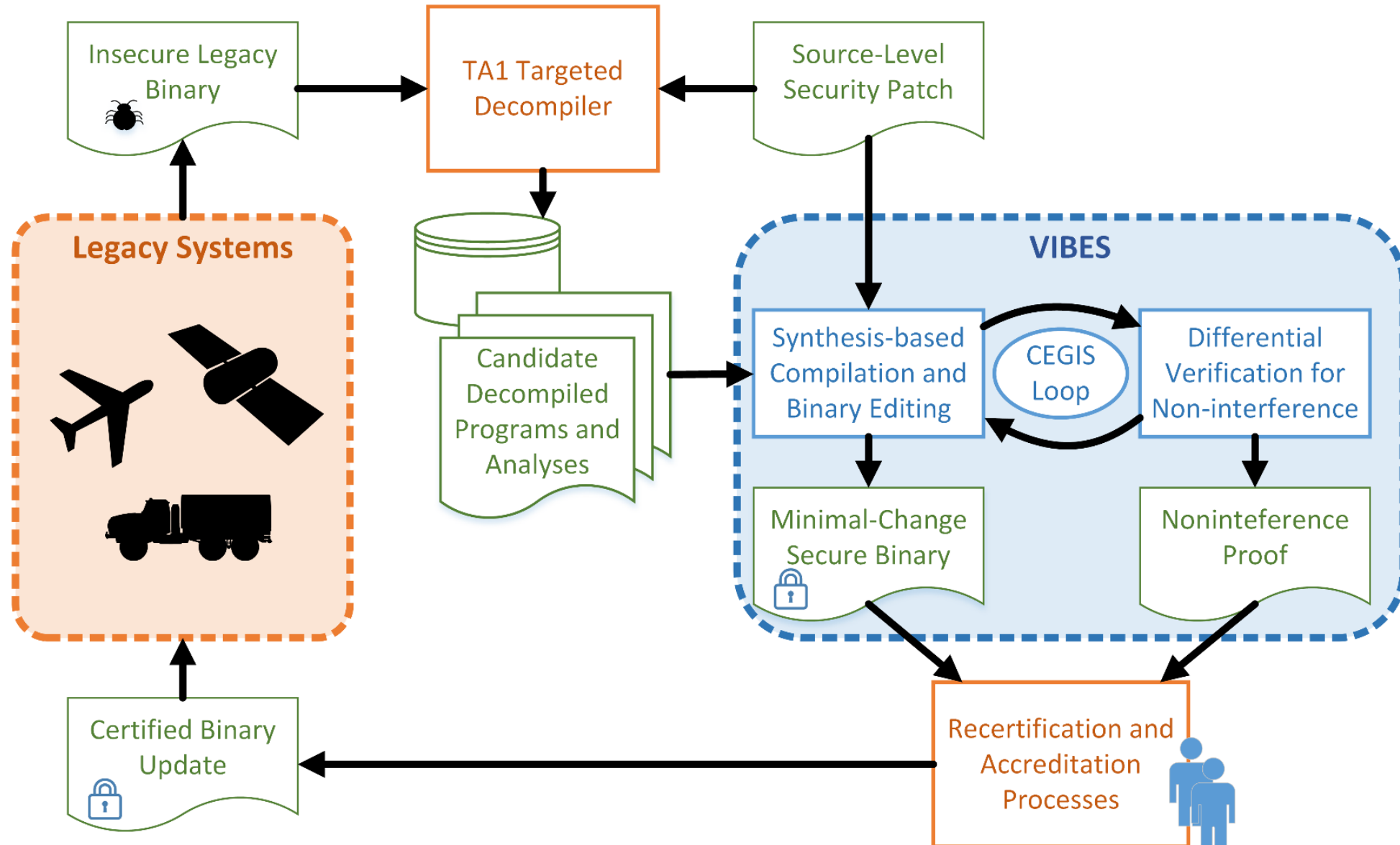
- Patching (Semi-automated approach)
  - Ability to semi-automatically “micropatch”
    - Dealing with identifying space in the original binary and “linking” the patch code with the original code
    - Preserved
    - Removal of the vulnerability
    - Performance (speed w.r.t. runtime requirements)
  - Static verification (potentially, human in the loop verification)
    - Ability to compute a patch’s semantic effects
    - Ability to visualize a patch’s semantic effects
  - Dynamic verification
    - Ability to emulate and instrument the original and the patch code in the different supported architecture
    - Ability to collect execution traces of the original and the patched binary
    - Ability to compare dynamically-collected traces, showing trace equivalency when the original/patched binaries are provided with benign inputs
    - Ability to estimate timing of emulated execution traces

What if there is no compiler?

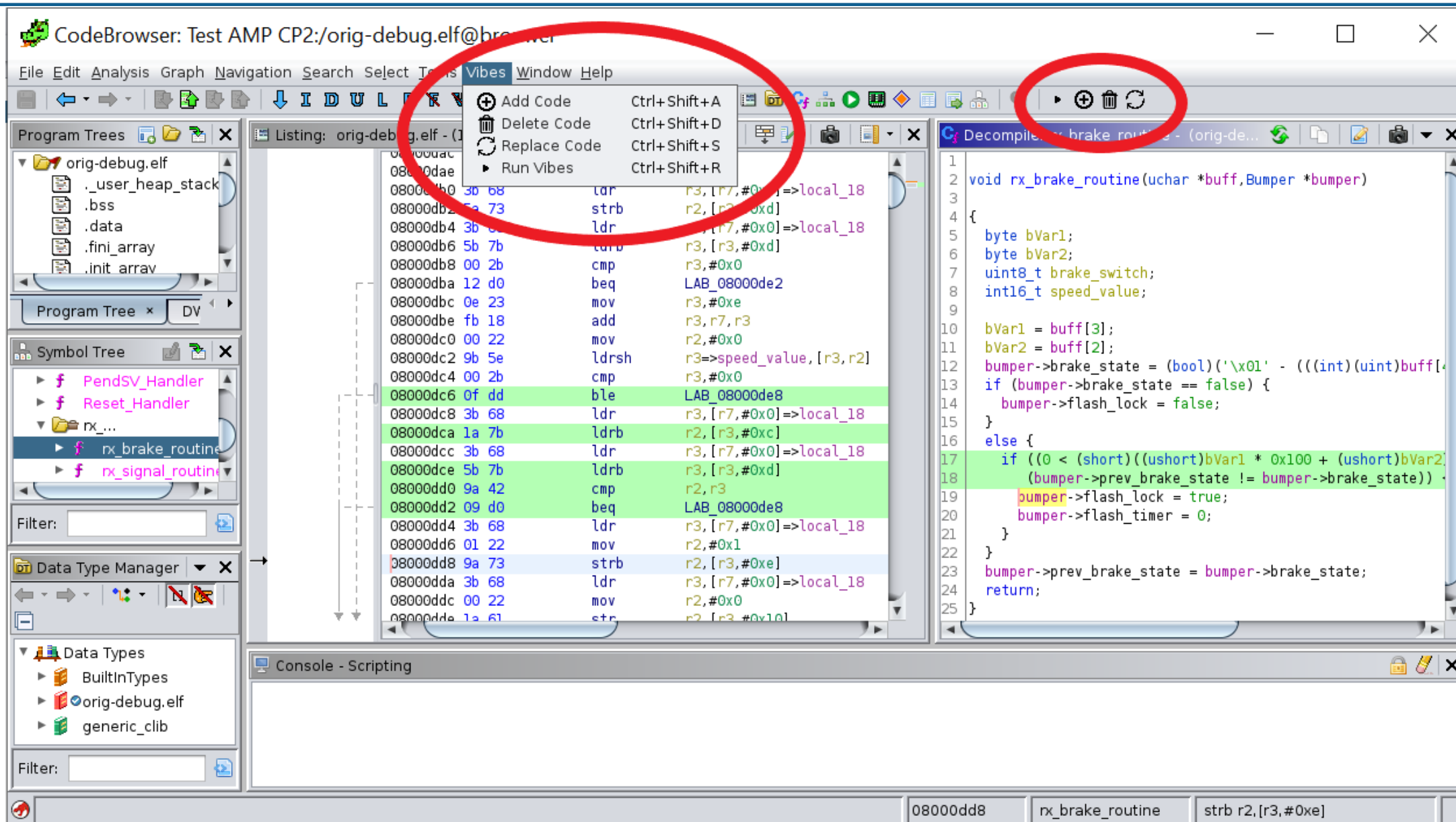
---

# Context and motivation for satellite patching challenge

## Verified, Incremental Binary Editing with Synthesis (VIBES) tool (Draper Labs)



# Goal: Usable front-end and AMP tool integration

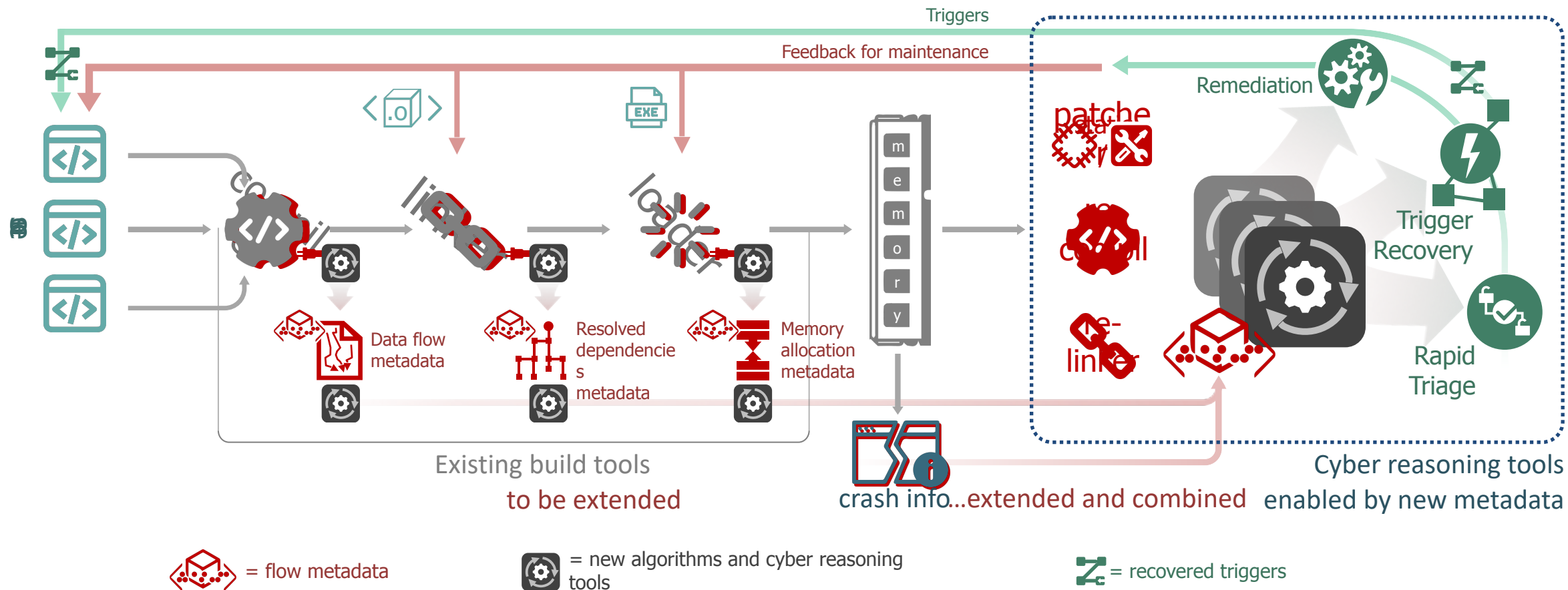




From maintaining individual firmware(s) to patching at scale

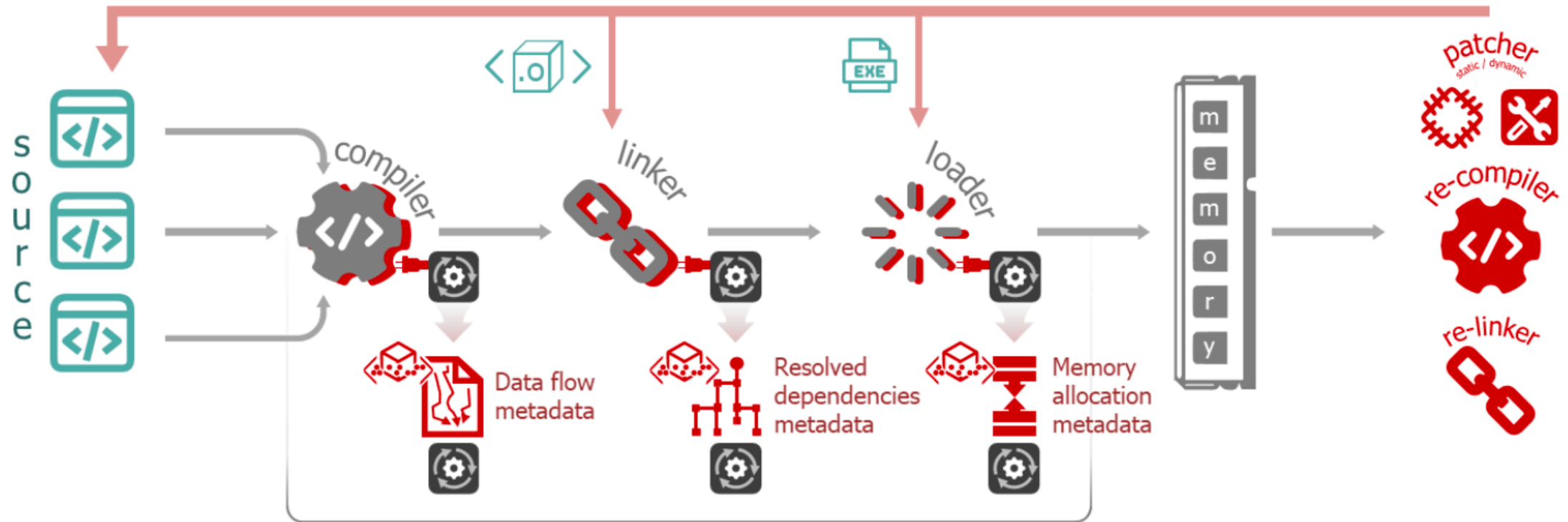
---

# E-BOSS: enhance SBOMs with flow metadata to trace flaws to triggers



- Keep advanced metadata in addition to symbols to effectively trace back flaw evidence to triggers
- Enhance SBOMs with new types of rich metadata, enabling cyber reasoning for triage and remediation
- Remediate with eSBOMs: Recover paths and triggers to crash site from crash snapshots ("crash dumps"), remediate by blocking triggers once recovered
  - Block triggers and flows leading to quick remediation

## New tools to maintain software post-compilation & post-linking



Advanced metadata is generated at each stage of the build process, enables maintenance of binaries



Thank you!



June 17, 2025

Hyatt Regency Crystal City, Arlington, VA

Save the Date

# Resilient Software Systems

COLLOQUIUM

Forging a New Era of Cyber Resiliency

There is never enough time. Thank you for yours!

---



## Appendix: DWARF links

---

- The Almighty DWARF: A Trojan Horse for Program Analysis, Verification, and Recompilation, Philip Zucker  
<https://www.philipzucker.com/dwarf-patching/>
- DWARF as a Shared Reverse Engineering Format, Romain Thomas,  
<https://lief.re/blog/2025-05-27-dwarf-editor/>