# Week 13: Lecture B
## Software Reverse Engineering

Thursday, November 20, 2025

# Announcements

- **Project 4: NetSec** released
  - **Deadline:** Thursday, December 4th by 11:59PM

## Project 4: Network Security

Deadline: **Thursday, December 4 by 11:59PM.**

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

### Helpful Resources

- The CS 4440 Course Wiki
- VM Setup and Troubleshooting
- Terminal Cheat Sheet

Table of Contents:


- Helpful Resources
- Introduction
- Objectives
- Start by reading this!
  - Packet Traces
  - Attack Template
  - Wireshark
- Part 1: Defending Networks
  - Password Cracking
  - Port Scanning
  - Anomalous Activity
  - What to Submit
- Part 2: Attacking Networks
  - Plaintext Credentials
  - Encoded Credentials
  - Accessed URLs
  - Extra Credit: Transferred Files
  - What to Submit
- Submission Instructions

# Interested in fuzzing?

- **Spring 2026: CS 5493/6493: Applied Software Security Testing**
  - **Everything you'd ever want to know about fuzzing for finding security bugs!**
  - Course project: team up to fuzz **a real program** (of your choice), and find and report its bugs!
  - http://cs.utah.edu/~snagy/courses/cs5493/

## CS 5493/6493: Applied Software Security Testing

This special topics course will dive into today's state-of-the-art techniques for uncovering hidden security vulnerabilities in software. Introductory fuzzing exercises will provide hands-on experience with industry-popular security tools such as AFL++ and AddressSanitizer, culminating in a final project where **you'll work to hunt down, analyze, and report security bugs in a real-world application of your choice**.

This class is open to graduate students and upper-level undergraduates. It is recommended you have a solid grasp over topics like software security, systems programming, and C/C++.

**Learning Outcomes:** At the end of the course, students will be able to:

- Design, implement, and deploy automated testing techniques to improve vulnerability on large and complex software systems.
- Assess the effectiveness of automated testing techniques and identify why they are well- or ill-suited to specific codebases.
- Distill testing outcomes into actionable remediation information for developers.
- Identify opportunities to adapt automated testing to emerging and/or unconventional classes of software or systems.
- Pinpoint testing obstacles and synthesize strategies to overcome them.
- Appreciate that testing underpins modern software quality assurance by discussing the advantages of proactive and post-deployment software testing efforts.

# Final Exam

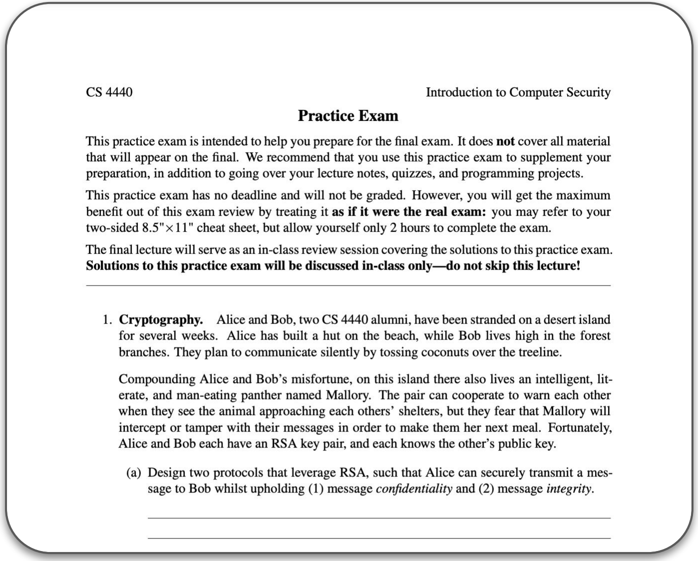
- **Save the date: 1–3PM** on **Wednesday, December 10**
  - **CDA accommodations:** schedule exam via CDA Portal

- **High-level details** (more to come):
  - One exam covering all course material
  - Similar to project/quiz/lecture exercises

- **Cheat Sheet**
  - **One 8.5"x11" paper** with handwritten/typed notes on **both** sides
  - **Suggestion:** Don't just use someone else's—you'll learn better making **your own**!
  - **Suggestion:** Don't just paste lecture slides—you'll learn better by **writing/typing** it!

# Practice Exam

- **Practice Exam** released
  - See **Assignments** page on the CS 4440 website
- **Final lecture** will serve as a **review session**
  - Solutions discussed **in-class only**—don't skip!



CS 4440 Introduction to Computer Security

**Practice Exam**

This practice exam is intended to help you prepare for the final exam. It does **not** cover all material that will appear on the final. We recommend that you use this practice exam to supplement your preparation, in addition to going over your lecture notes, quizzes, and programming projects.

This practice exam has no deadline and will not be graded. However, you will get the maximum benefit out of this exam review by treating it **as if it were the real exam:** you may refer to your two-sided 8.5"×11" cheat sheet, but allow yourself only 2 hours to complete the exam.

The final lecture will serve as an in-class review session covering the solutions to this practice exam. **Solutions to this practice exam will be discussed in-class only—do not skip this lecture!**

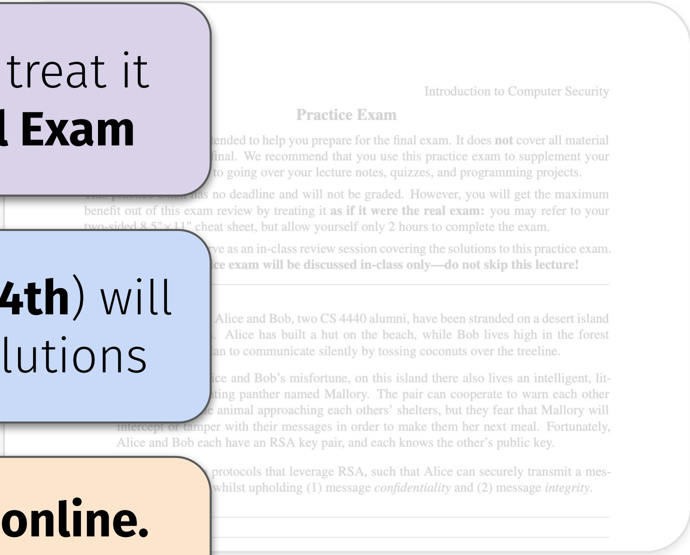
1. **Cryptography.** Alice and Bob, two CS 4440 alumni, have been stranded on a desert island for several weeks. Alice has built a hut on the beach, while Bob lives high in the forest branches. They plan to communicate silently by tossing coconuts over the treeline.

   Compounding Alice and Bob's misfortune, on this island there also lives an intelligent, literate, and man-eating panther named Mallory. The pair can cooperate to warn each other when they see the animal approaching each others' shelters, but they fear that Mallory will intercept or tamper with their messages in order to make them her next meal. Fortunately, Alice and Bob each have an RSA key pair, and each knows the other's public key.

   (a) Design two protocols that leverage RSA, such that Alice can securely transmit a message to Bob whilst upholding (1) message *confidentiality* and (2) message *integrity*.

# Practice Exam

- Practice Exam re
  - See Assignmen
- Final lecture will serve as a review session
  - Solutions discu




To get the most out of this, treat it just **as you would the Final Exam**

Last lecture (**Thursday, Dec. 4th**) will go over the exam review solutions

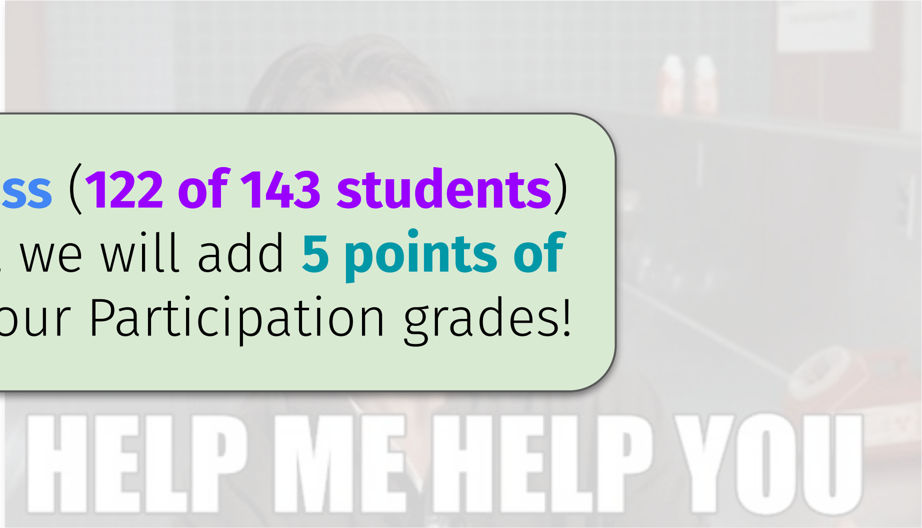
**Solutions won't be posted online.**
(Reminder: attendance/participation makes up **5%** of your course grade)

# End-of-semester Course Evals

- **I want your feedback!**
  - 3rd time teaching this course 😃
  - **Help me improve the class!**
- Due by **December 15th**
  - https://scf.utah.edu
  - **Please please please!**

# End-of-semester Course Evals

- **I want your feedback!**
  - 3rd time teaching this course 😃
  - **Help me improve the class!**
- Due by **Dec**
  - https://sc
  - **Please pl**




**If 85% of the class** (**122 of 143 students**) submits an eval, we will add **5 points of extra credit** to your Participation grades!

# Questions?

# No Class or Office Hours Next Week

# Last time on CS 4440...

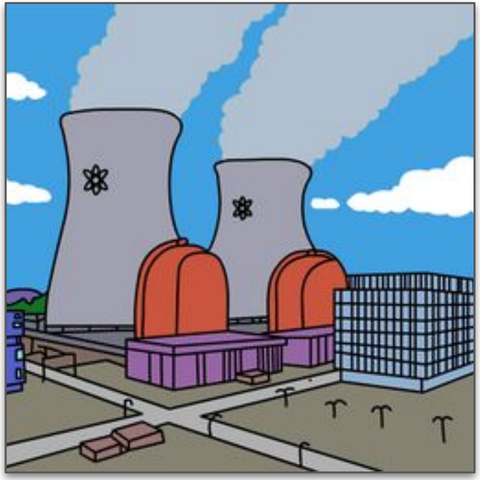
Cyber-physical Systems &
Internet-of-Things Security

# Cyber-physical Systems

- **What Cyber-physical Systems do you directly/indirectly interact with daily?**

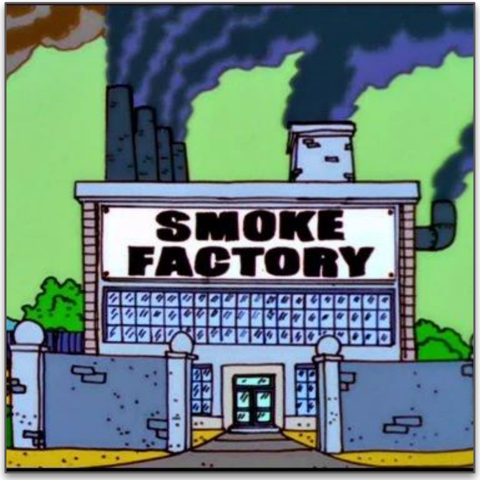
# Cyber-physical Systems

- **What Cyber-physical Systems do you directly/indirectly interact with daily?**



Power Grid

Telecommunication



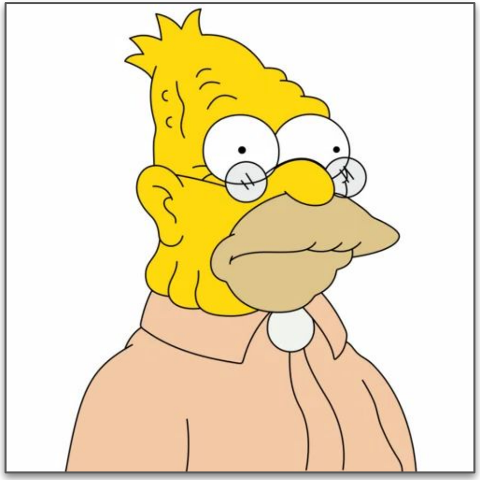

Critical Industry

# Cyber-physical Systems

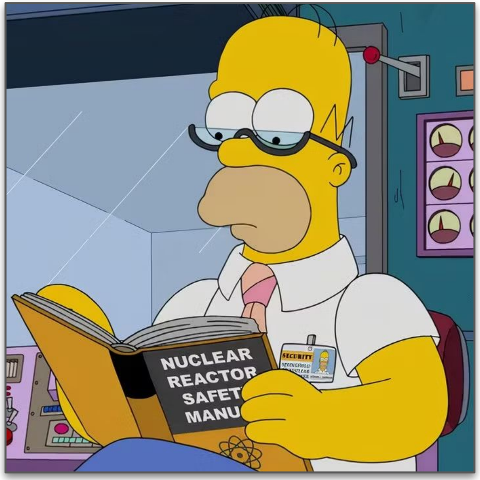
- **Why are Cyber-physical Systems so challenging to defend?**

# Cyber-physical Systems

- **Why are Cyber-physical Systems so challenging to defend?**



Legacy Components

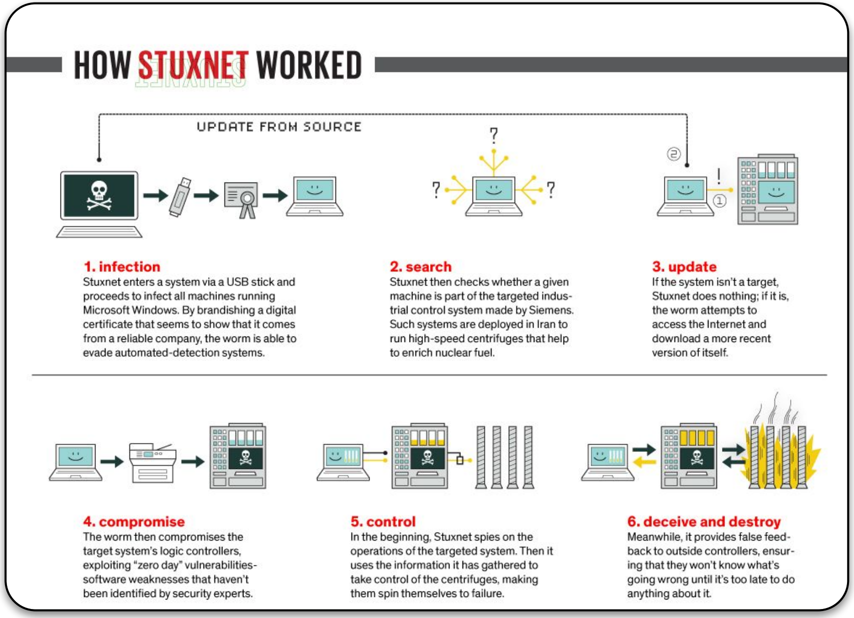



Unpatchable Systems


Stealthy Attacks

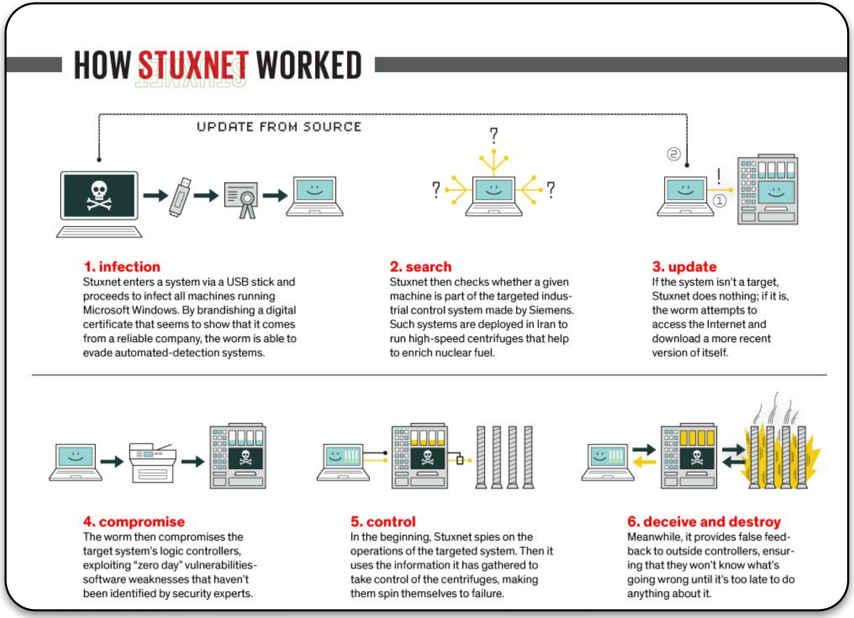
# CPS Attacks are Here to Stay

- **2010's Stuxnet attack**



# CPS Attacks are Here to Stay

- 2010's Stuxnet attack... was just the tip of the iceberg

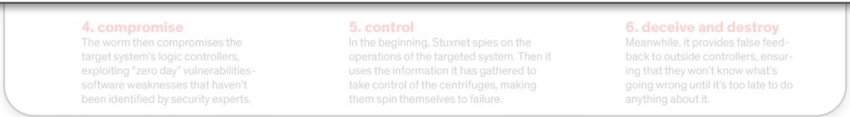


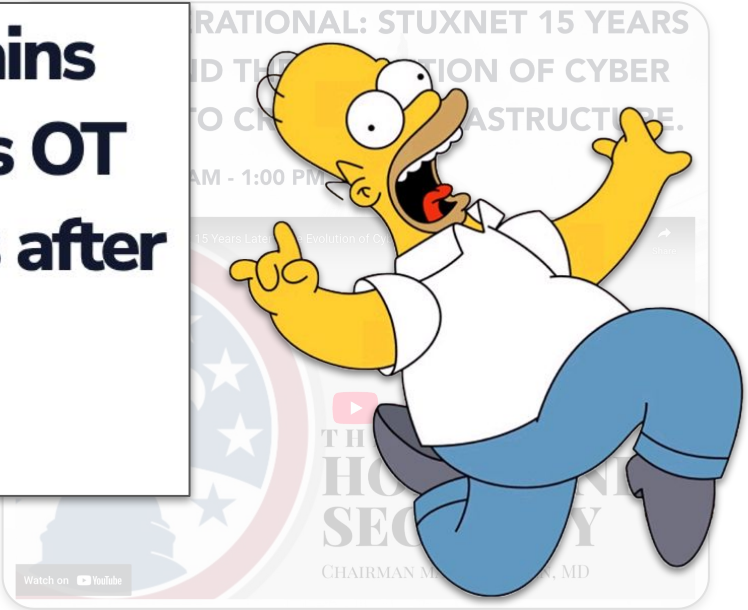

# CPS Attacks are Here to Stay

- 2010's Stuxnet attack… was just the tip of the iceberg

**US critical infrastructure remains exposed as Congress confronts OT cybersecurity gaps, fifteen years after Stuxnet**

JULY 22, 2025



# Forthcoming Course Offerings

- **Prof. Garcia's Course returning Fall 2026**
  - Note course number will now be **CS 5464**

SYLLABUS

CS 6963/5963: CYBER-PHYSICAL SYSTEMS (CPS) AND INTERNET-OF-THINGS (IOT) SECURITY

INSTRUCTOR

**Instructor:** Luis Garcia

**Department:** CS

**Office:** MEB 3450

**E-mail:** la.garcia@utah.edu

**Communication & Office Hours:** Review the "Communication" section below for more information.

**Teaching Assistant:** Vatsal Goel

**Email:** vatsal.goel@utah.edu

**Pre-requisites:** CS 3505

**Credit Hours:** 3.0

**Semester:** Fall 2024

**PDF Syllabus:** N/A

# Questions?

# This time on CS 4440…

Binary Reverse Engineering
Instruction Recovery
Control Flow Analysis
Structure Recovery
RE Challenges

# About Me

- **Zao Yang**
- 3rd year PhD student advised by Prof. Nagy
- Research areas: fuzzing, binary analysis (today's topic!)
- Contact: zao.yang@utah.edu

# How Software is Built

- `clang hello.c -o hello`

# How Software is Built

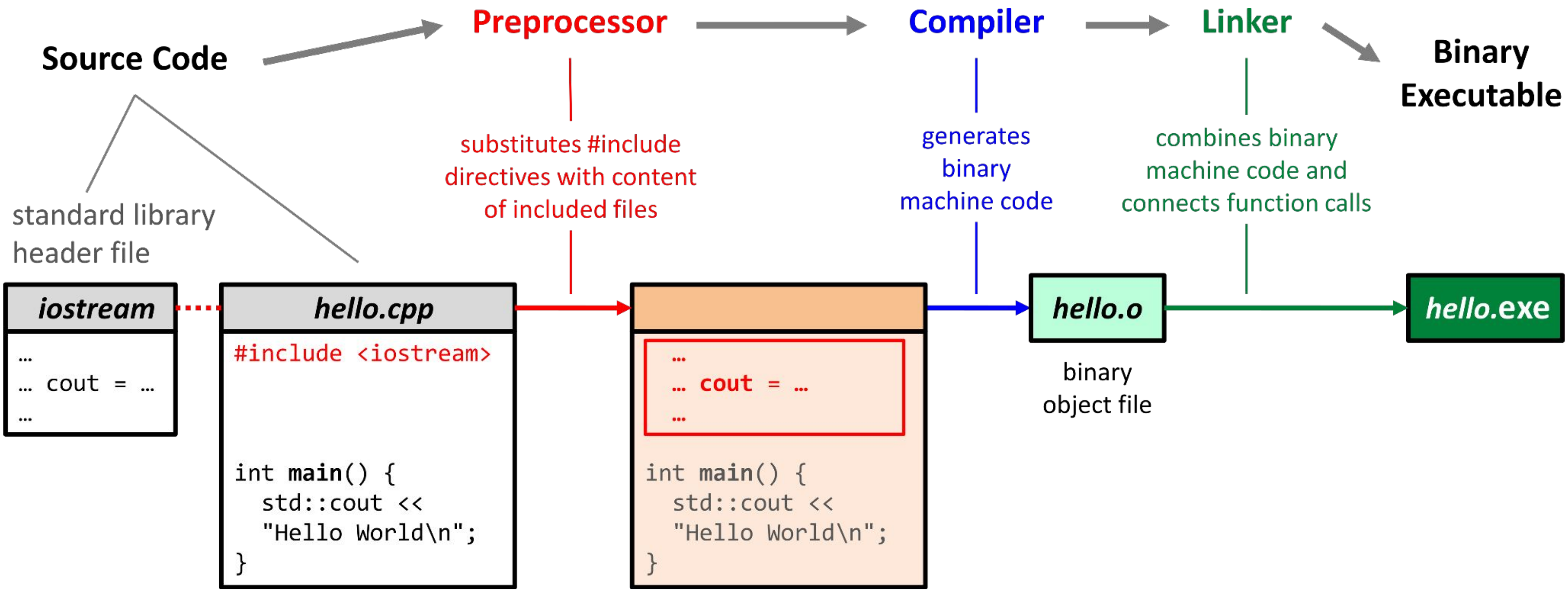
- `clang hello.c -o hello`

Compiler

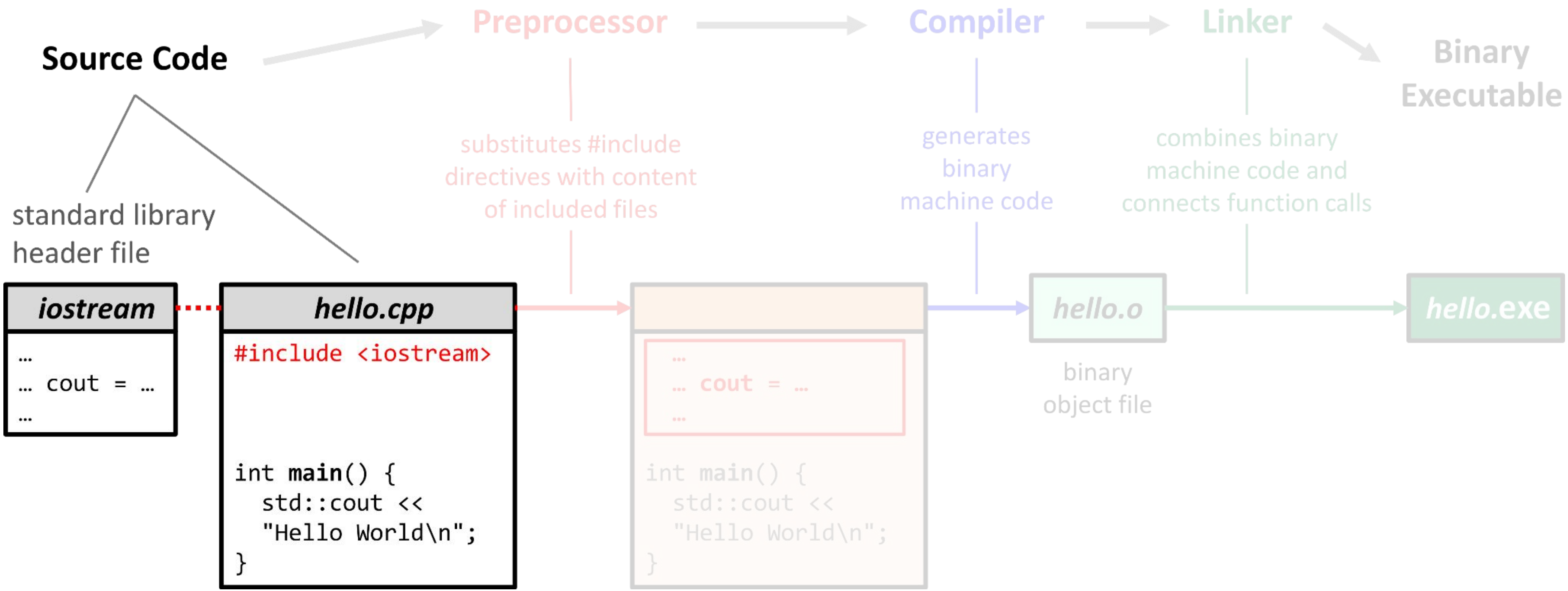
Source File

Executable

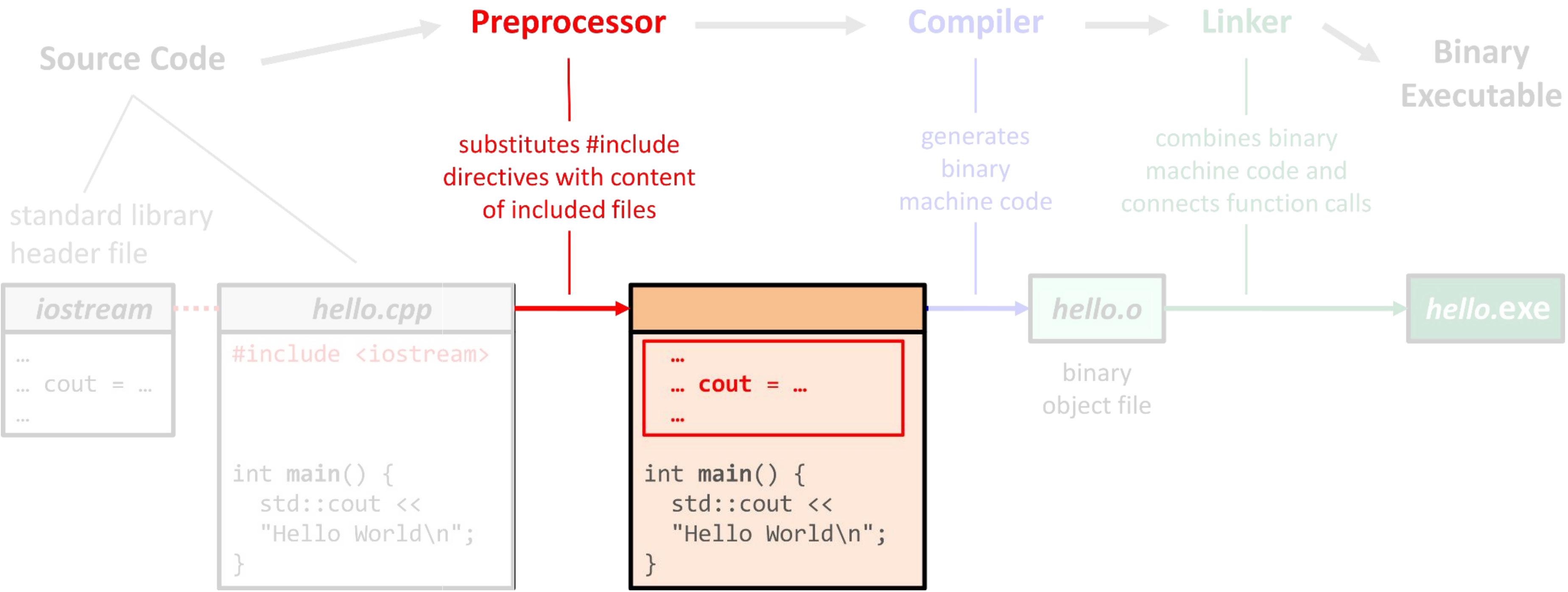
# How Software is Built



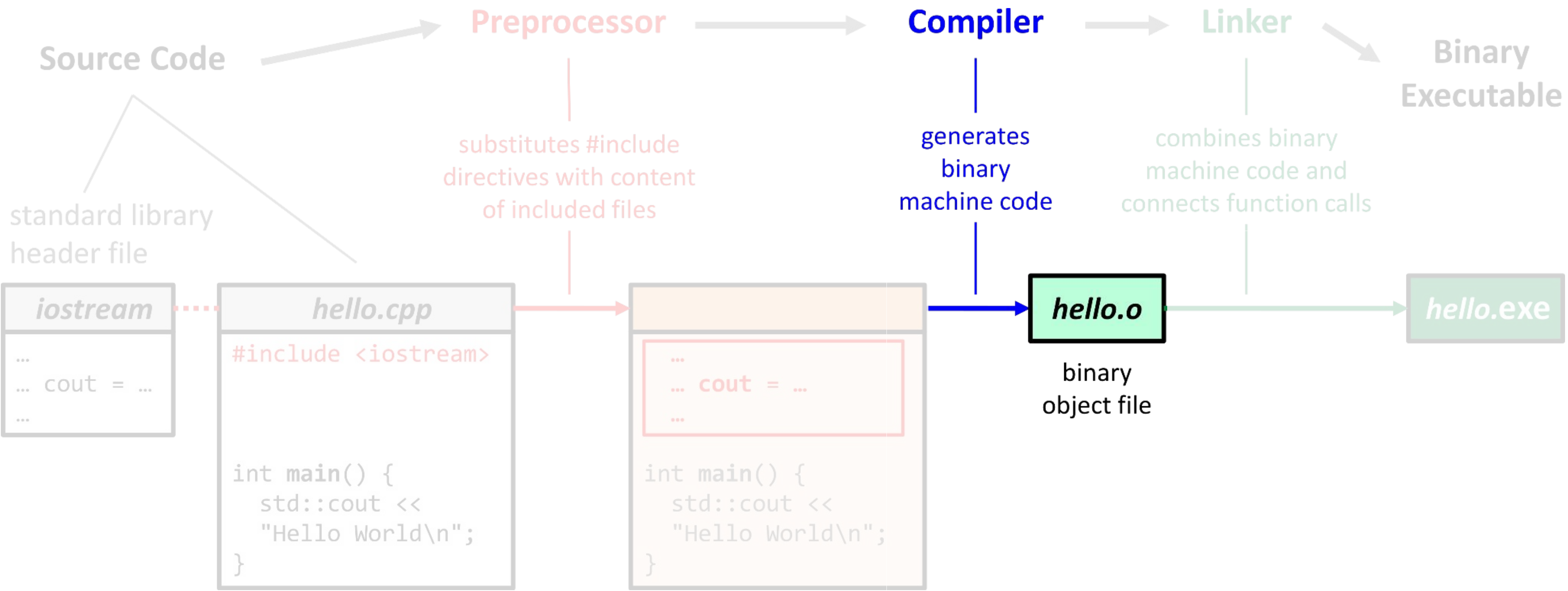
# How Software is Built



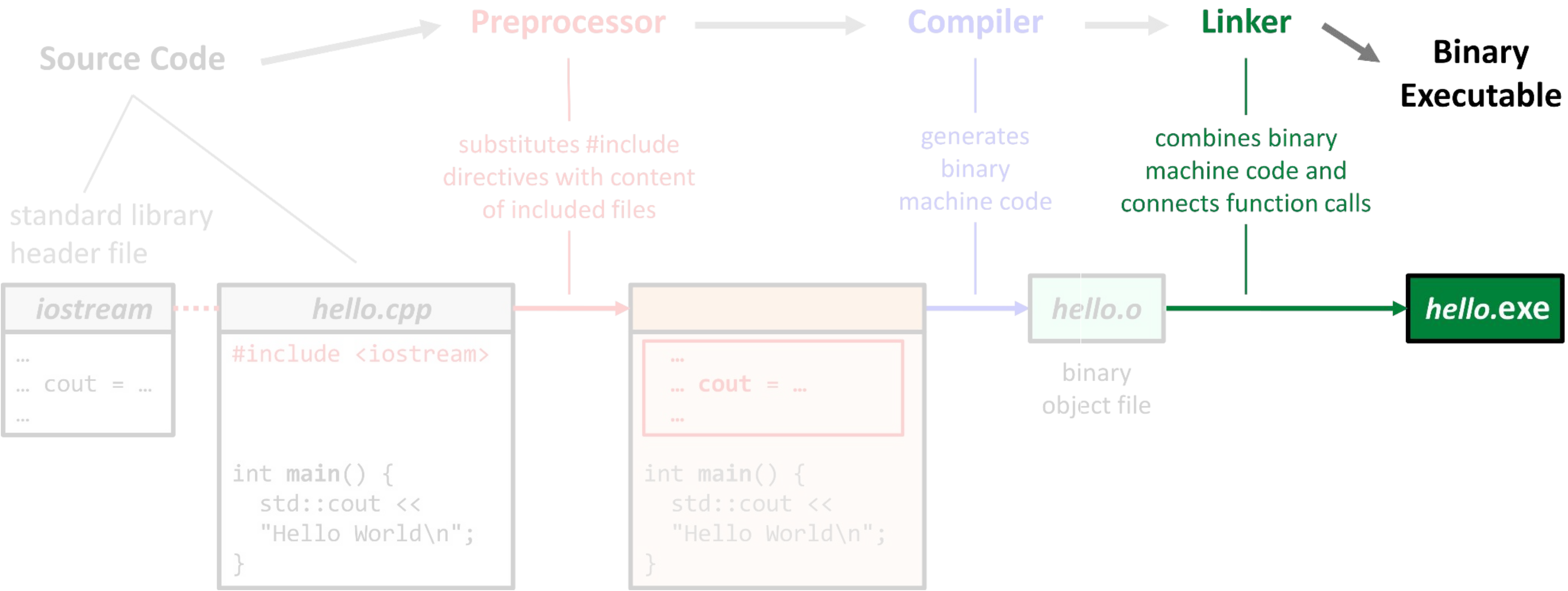
# How Software is Built



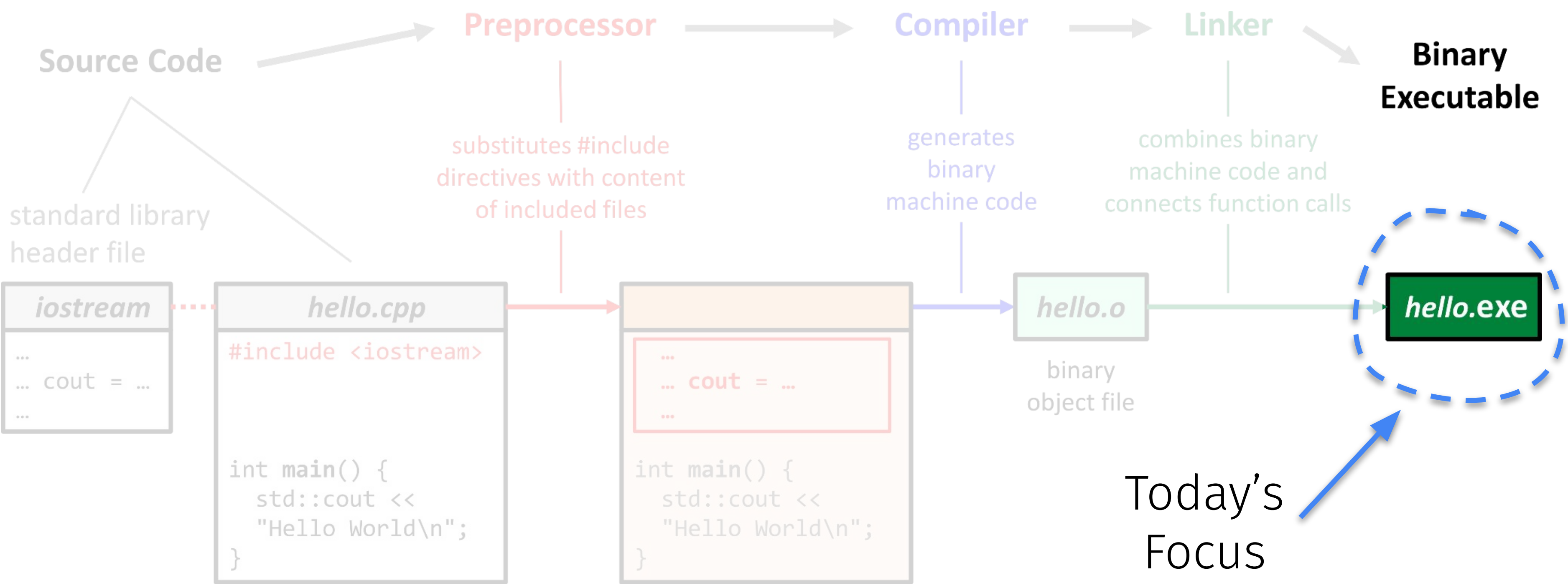
# How Software is Built



# How Software is Built



# How Software is Built



# Closed-source Software

- **Examples?**

skype Office NVIDIA

Dropbox Outlook

Windows iOS macOS cisco

ORACLE Solaris PlayStation NETGEAR

# Closed-source Software

- Examples?

**Commercialized** applications and libraries

Freely-distributed **proprietary software**

**Legacy software** whose source code is lost

# Auditing Open- versus Closed-source Code

Open Source:

- **Publicly-available** source codebase
- Achieves security by **transparency**


- **Semantic richness** facilitates **high-performance**, **effective** vetting

# Auditing Open- versus Closed-source Code

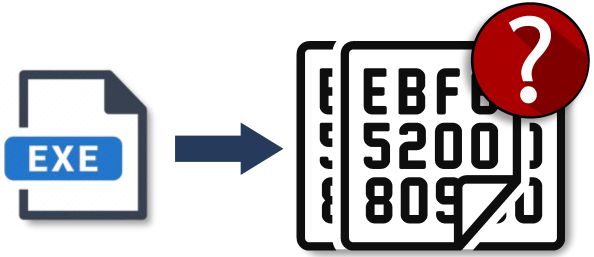
## Open Source:

- **Publicly-available** source codebase
- Achieves security by **transparency**

- **Semantic richness** facilitates **high-performance**, **effective** vetting

## Closed Source:

- Distributed as a precompiled **binary**
- **Opaque to *everyone*** but its developer



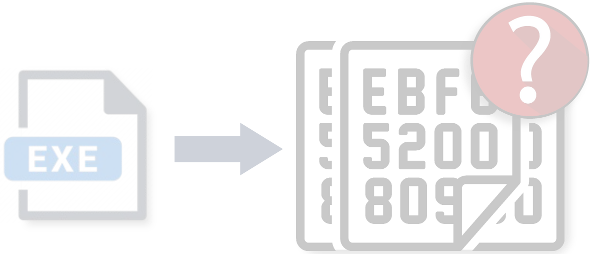

- **Upwards of 10x slower** security vetting
- Forced to **rely on crude techniques**

# Auditing Open- versus Closed-source Code

- Global market size **over $240 billion**
- **85%** contains **critical vulnerabilities**
- **89%** of the **most exploited software**

## Closed Source:

- Distributed as a precompiled **binary**
- **Opaque to *everyone*** but its developer




- **Upwards of 10x slower** security vetting
- Forced to **rely on crude techniques**

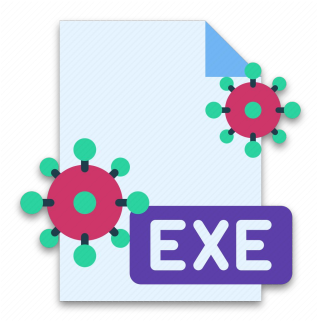
# Reverse Engineering (RE)

- **What is RE?**

  "A process or method through which one attempts to **understand** through deductive reasoning how a previously made **device**, **process**, **system**, or piece of **software** accomplishes a task with **very little (if any) insight** into exactly how it does so."

# Why do we care about RE?

- **Discovering bugs**
- **Retrofitting fixes**
- **Malware analysis**
- **Right to repair!**




# RE Tasks

- **Disassembly**
  - **???**

# RE Tasks

- **Disassembly**
  - Machine code to human readable assembly
- **Decompilation**
  - ???

```
00000003  4F              dec edi
00000004  6A1E            push byte +0x1e
00000006  B7B5            mov bh,0xb5
00000008  0C12            or al,0x12
0000000A  6A04            push byte +0x4
0000000C  EAA08EA57B2BB1  jmp dword 0xb12b:0x7ba58ea0
00000013  B114            mov cl,0x14
```

# RE Tasks

- **Disassembly**
  - Machine code to human readable assembly
- **Decompilation**
  - Machine code to human readable source code
- **Rewriting**
  - ???

| | | |
|---|---|---|
| 00000003 | 4F | dec edi |
| 00000004 | 6A1E | push byte +0x1e |
| 00000006 | B7B5 | mov bh,0xb5 |
| 00000008 | 0C12 | or al,0x12 |
| 0000000A | 6A04 | push byte +0x4 |
| 0000000C | EAA08EA57B2BB1 | jmp dword 0xb12b:0x7ba58ea0 |
| 00000013 | B114 | mov cl,0x14 |

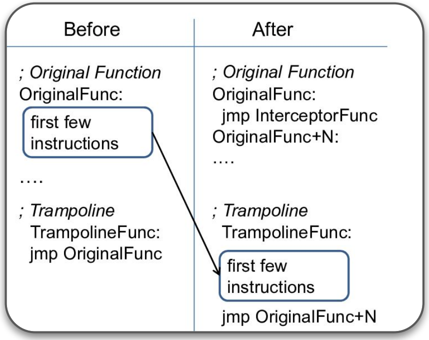
```
int main(void)

{
  char local_54 [64];
  int modified;

  modified = 0;
  gets(local_54);
  if (modified == 0) {
    puts("Try again?");
  }
  else {
    puts("you have changed the \'modified\' variable");
  }
  return 0;
}
```

# RE Tasks

- **Disassembly**
  - Machine code to human readable assembly
- **Decompilation**
  - Machine code to human readable source code
- **Rewriting**
  - Add more functionality and rebuild executable

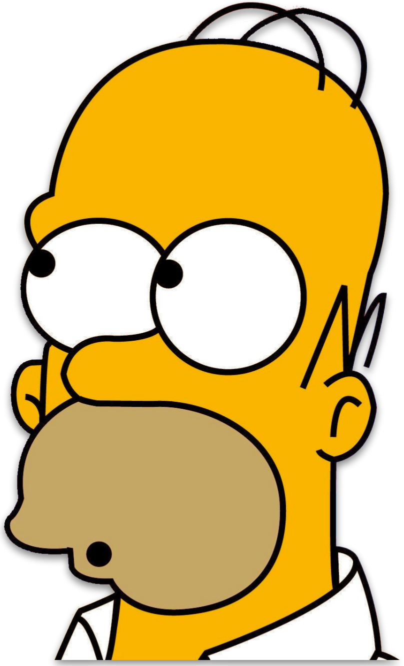
```
00000003  4F             dec edi
00000004  6A1E           push byte +0x1e
00000006  B7B5           mov bh,0xb5
00000008  0C12           or al,0x12
0000000A  6A04           push byte +0x4
0000000C  EAA08EA57B2BB1 jmp dword 0xb12b:0x7ba58ea0
00000013  B114           mov cl,0x14
```

```c
int main(void)

{
  char local_54 [64];
  int modified;

  modified = 0;
  gets(local_54);
  if (modified == 0) {
    puts("Try again?");
  }
  else {
    puts("you have changed the \'modified\' variable");
  }
  return 0;
}

```



# Three Pillars of RE

1. **Instruction Recovery**
   - Decode bytes to instructions
   - Disambiguate code from data
2. **Control Flow Recovery**
   - Intra-procedural execution flow
   - Inter-procedural execution flow
3. **Program Structure Recovery**
   - Identify program basic blocks
   - Higher-level constructs (e.g., loops)



# Questions?

# Pillars of RE:
# Instruction Recovery

# Instructions

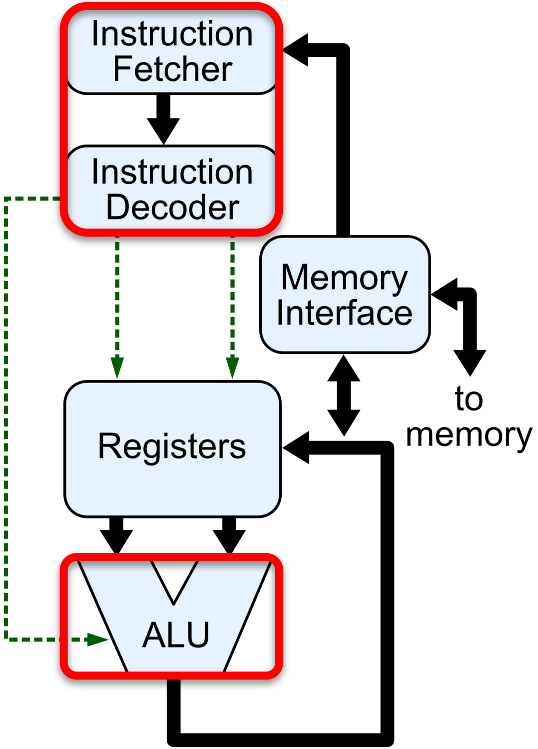
- **What are they?**

| | |
|---|---|
| MAXPS | Maximum of Packed Single-Precision Floating-Point Values |
| MAXSD | Return Maximum Scalar Double-Precision Floating-Point Value |
| MAXSS | Return Maximum Scalar Single-Precision Floating-Point Value |
| MFENCE | Memory Fence |
| MINPD | Minimum of Packed Double-Precision Floating-Point Values |
| MINPS | Minimum of Packed Single-Precision Floating-Point Values |
| MINSD | Return Minimum Scalar Double-Precision Floating-Point Value |
| MINSS | Return Minimum Scalar Single-Precision Floating-Point Value |
| MONITOR | Set Up Monitor Address |
| MOV | Move |
| MOV (1) | Move to/from Control Registers |
| MOV (2) | Move to/from Debug Registers |
| MOVAPD | Move Aligned Packed Double-Precision Floating-Point Values |
| MOVAPS | Move Aligned Packed Single-Precision Floating-Point Values |
| MOVBE | Move Data After Swapping Bytes |
| MOVD | Move Doubleword/Move Quadword |
| MOVDDUP | Replicate Double FP Values |
| MOVDIR64B | Move 64 Bytes as Direct Store |
| MOVDIRI | Move Doubleword as Direct Store |
| MOVDQ2Q | Move Quadword from XMM to MMX Technology Register |

# Recap: The CPU

- State modified by assembly **instructions**
  - ADD, SUB, XOR, CMP, CALL, JMP, RET
  - **And many more!**

- Assembly instruction syntaxes
  - **AT&T** = `Instruction Source Destination`
  - **Intel** = `Instruction Destination Source`
  - Example: **MOV** SRC, DST versus **MOV** DST, SRC
  - This lecture: **AT&T syntax**



# Instructions

- **What are they?**
  - Operations that modify **CPU state**
- Source = **???**
- x86 asm = **???**

| | |
|---|---|
| MAXPS | Maximum of Packed Single-Precision Floating-Point Values |
| MAXSD | Return Maximum Scalar Double-Precision Floating-Point Value |
| MAXSS | Return Maximum Scalar Single-Precision Floating-Point Value |
| MFENCE | Memory Fence |
| MINPD | Minimum of Packed Double-Precision Floating-Point Values |
| MINPS | Minimum of Packed Single-Precision Floating-Point Values |
| MINSD | Return Minimum Scalar Double-Precision Floating-Point Value |
| MINSS | Return Minimum Scalar Single-Precision Floating-Point Value |
| MONITOR | Set Up Monitor Address |
| MOV | Move |
| MOV (1) | Move to/from Control Registers |
| MOV (2) | Move to/from Debug Registers |
| MOVAPD | Move Aligned Packed Double-Precision Floating-Point Values |
| MOVAPS | Move Aligned Packed Single-Precision Floating-Point Values |
| MOVBE | Move Data After Swapping Bytes |
| MOVD | Move Doubleword/Move Quadword |
| MOVDDUP | Replicate Double FP Values |
| MOVDIR64B | Move 64 Bytes as Direct Store |
| MOVDIRI | Move Doubleword as Direct Store |
| MOVDQ2Q | Move Quadword from XMM to MMX Technology Register |

# Instructions

- **What are they?**
  - Operations that modify **CPU state**
- Source = **high-level** instructions
  - Human-readable
- x86 asm = **low-level** instructions
  - Somewhat human-readable

Key to inferring what the program is doing

| | |
|---|---|
| MAXPS | Maximum of Packed Single-Precision Floating-Point Values |
| MAXSD | Return Maximum Scalar Double-Precision Floating-Point Value |
| MAXSS | Return Maximum Scalar Single-Precision Floating-Point Value |
| MFENCE | Memory Fence |
| MINPD | Minimum of Packed Double-Precision Floating-Point Values |
| MINPS | Minimum of Packed Single-Precision Floating-Point Values |
| MINSD | Return Minimum Scalar Double-Precision Floating-Point Value |
| MINSS | Return Minimum Scalar Single-Precision Floating-Point Value |
| MONITOR | Set Up Monitor Address |
| MOV | Move |
| MOV (1) | Move to/from Control Registers |
| MOV (2) | Move to/from Debug Registers |
| MOVAPD | Move Aligned Packed Double-Precision Floating-Point Values |
| MOVAPS | Move Aligned Packed Single-Precision Floating-Point Values |
| MOVBE | Move Data After Swapping Bytes |
| MOVD | Move Doubleword/Move Quadword |
| MOVDDUP | Replicate Double FP Values |
| MOVDIR64B | Move 64 Bytes as Direct Store |
| MOVDIRI | Move Doubleword as Direct Store |
| MOVDQ2Q | Move Quadword from XMM to MMX Technology Register |

# Recovering Instructions

- **Goal:** translate bytes into **logical instructions**
  - Called instruction **decoding**
  - Analogous to what CPU does
  - General output: **disassembly**

Instruction stream

```
B8 22 11 00 FF 01 CA 31 F6 53 8B 5C 24
04 8D 34 48 39 C3 72 EB C3
```

Read bytes from input executable

Machine code bytes

```
B8 22 11 00 FF
01 CA
31 F6
53
8B 5C 24 04
8D 34 48
39 C3
72 EB
C3
```

Group bytes

Assembly language statements

```
foo:
movl $0xFF001122, %eax
addl %ecx, %edx
xorl %esi, %esi
pushl %ebx
movl 4(%esp), %ebx
leal (%eax,%ecx,2), %esi
cmpl %eax, %ebx
jnae foo
retl
```

Decode instructions

# Instruction Recovery Techniques

- **Linear Sweep**
  - Start decoding at binary entry
  - Attempt to decode all bytes
  - Stop at end of .TEXT section

**Intuition:** compilers lay code sequentially for compactness

**Challenge:** data within code

# Instruction Recovery Techniques

- **Linear Sweep**
  - Start decoding at binary entry
  - Attempt to decode all bytes
  - Stop at end of .TEXT section

**Intuition:** compilers lay code sequentially for compactness

**Challenge:** data within code

- **Recursive Descent**
  - Follow all control-flow transfers
  - `jmp 0x100` → start decoding instructions at address `0x100`
  - Stop when you've covered all possible control-flow paths

**Intuition:** following the logical flow of execution reveals a lot

**Challenge:** indirect branches

# Instruction Recovery Techniques

- **Linear Sweep**
  - Start decoding at binary entry
  - Attempt to decode all bytes
  - Stop at en

**Intuition:** compilers lay code sequentially for compactness

ithin code

- **Recursive D**
  - Follow all
  - `jmp 0x100` → start decoding instructions at address `0x100`
  - Stop when you've covered all possible control-flow paths

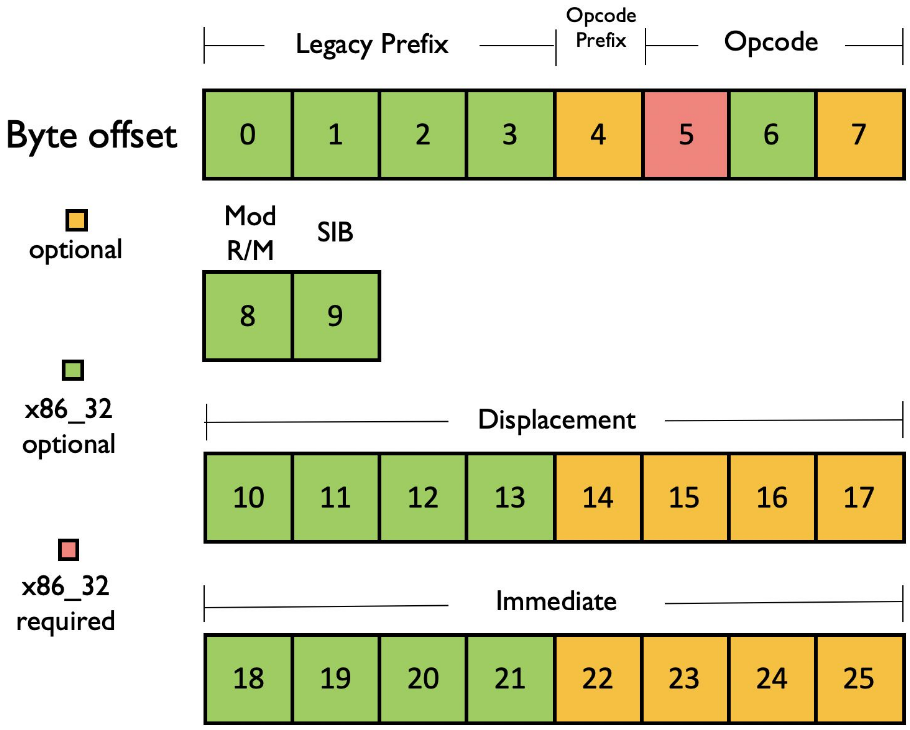
g the logical flow of execution reveals a lot

**Challenge:** indirect branches

**Most modern RE** adopts a **combined** approach in addition to **heuristics**

# CISC Architectures

- **Variable-length** instructions
  - E.g., x86-32, x86-64
- Almost any byte sequence can be a **valid instruction**!
- Being just one byte off can totally mess up decoding!



# CISC Architectures

- Example of byte offsets and possible decodings:

```
0x0F 0x88 0x52 0x0F 0x84 0xEC
```

```
js      0xffffffffec840f58
```

# CISC Architectures

- Example of byte offsets and possible decodings:

```
0x0F 0x88 0x52 0x0F 0x84 0xEC
```

```
mov     BYTE PTR [rdx+0xf],dl
test    ah,ch
```

# CISC Architectures

- Example of byte offsets and possible decodings:

```
0x0F 0x88 0x52 0x0F 0x84 0xEC
```

```
add    eax,0x40080f20
in     al,dx
```

# Instruction Decoder Bugs

- Results from Trail of Bits' Mishegos fuzzer:

| input | worker | | | |
|---|---|---|---|---|
| | ./src/worker/bfd/bfd.so | ./src/worker/capstone/capstone.so | ./src/worker/xed/xed.so | ./src/worker/zydis/zydis.so |
| 6567f2f39cb2a58654 | gs addr32 repnz repz pushf (5 / 28) | pushfq (5 / 8) | addr32 pushfq (5 / 14) | pushfq (5 / 6) |
| 26f267664d0f3817314aecdf9d | es addr32 data16 rex.WRB (bad) (8 / 32) | ptest xmm14, xmmword ptr es:[r9d] (9 / 34) | (0 / 0) | (0 / 0) |
| 656636f26f0f3a6f959066b1fd8c52 | gs repnz outs dx,WORD PTR ss:[rsi] (5 / 35) | repne outsd dx, dword ptr ss:[rsi] (5 / 35) | repne data16 outsw gs (5 / 21) | repne outsw (5 / 11) |
| 67f03ef0460f1104fe | lock lock movups XMMWORD PTR ds:[esi+r15d*8],xmm8 (9 / 50) | (0 / 0) | (0 / 0) | (0 / 0) |
| 652e26520ffda71fd5bc9e3090235f | gs cs es push rdx (4 / 18) | push rdx (4 / 9) | push rdx (4 / 8) | push rdx (4 / 8) |
| 64652e26520ffda71fd5bc9e309023 | fs gs cs es push rdx (5 / 21) | push rdx (5 / 9) | push rdx (5 / 8) | push rdx (5 / 8) |
| 6636f26f0f3a6f959066b1fd8c523e | repnz outs dx,WORD PTR ss:[rsi] (4 / 32) | repne outsd dx, dword ptr ss:[rsi] (4 / 35) | repne data16 outsw (4 / 19) | repne outsw (4 / 11) |
| 2e26520ffda71fd5bc9e3090235f0d | cs es push rdx (3 / 15) | push rdx (3 / 9) | push rdx (3 / 8) | push rdx (3 / 8) |
| 36f2f3f00f3a6315cd | ss repnz lock (bad) (7 / 21) | (0 / 0) | (0 / 0) | (0 / 0) |
| 262ef0f00f3ada4eb5bae8 | es cs lock lock (bad) (7 / 23) | (0 / 0) | (0 / 0) | (0 / 0) |
| f365f33e9559dd95a732b088057275 | repz gs repz ds xchg ebp,eax (5 / 29) | xchg eax, ebp (5 / 14) | xchg ebp, eax (5 / 13) | xchg ebp, eax (5 / 13) |
| 65f33e9559dd95a732b08805727509 | gs repz ds xchg ebp,eax (4 / 24) | xchg eax, ebp (4 / 14) | xchg ebp, eax (4 / 13) | xchg ebp, eax (4 / 13) |
| f33e9559dd95a732b08805727509f3 | repz ds xchg ebp,eax (3 / 21) | xchg eax, ebp (3 / 14) | xchg ebp, eax (3 / 13) | xchg ebp, eax (3 / 13) |
| 3e9559dd95a732b08805727509f395 | ds xchg ebp,eax (2 / 16) | xchg eax, ebp (2 / 14) | xchg ebp, eax (2 / 13) | xchg ebp, eax (2 / 13) |
| 6765676547be4b69 | addr32 (1 / 7) | (0 / 0) | (0 / 0) | (0 / 0) |
| 59dd95a732b08805727509f39507d0 | pop rcx (1 / 11) | pop rcx (1 / 8) | pop rcx (1 / 7) | pop rcx (1 / 7) |
| 3e3e2e265ddf3b | ds ds cs es pop rbp (5 / 20) | pop rbp (5 / 8) | pop rbp (5 / 7) | pop rbp (5 / 7) |
| 9559dd95a732b08805727509f39507 | xchg ebp,eax (1 / 15) | xchg eax, ebp (1 / 14) | xchg ebp, eax (1 / 13) | xchg ebp, eax (1 / 13) |
| 646526f04991a85e | fs gs es lock xchg r9,rax (6 / 26) | (0 / 0) | (0 / 0) | (0 / 0) |
| f26426644a0f38d5fbbe | repnz fs es fs rex.WX (bad) (8 / 29) | (0 / 0) | (0 / 0) | (0 / 0) |
| 64666566950f3a13f9f6 | fs data16 gs xchg bp,ax (5 / 24) | xchg ax, bp (5 / 12) | xchg bp, ax (5 / 11) | xchg bp, ax (5 / 11) |
| f02ef065440f5a52e209f49611d758 | lock cs lock cvtps2pd xmm10,QWORD PTR gs:[rdx-0x1e] (9 / 52) | (0 / 0) | (0 / 0) | (0 / 0) |
| 36266467470f3875ae022de4a1517e | ss es fs addr32 rex.RXB (bad) (8 / 31) | (0 / 0) | (0 / 0) | (0 / 0) |
| 266467470f3875ae022de4a1517e90 | es fs addr32 rex.RXB (bad) (7 / 28) | (0 / 0) | (0 / 0) | (0 / 0) |
| 6467470f3875ae022de4a1517e90b4 | fs addr32 rex.RXB (bad) (6 / 25) | (0 / 0) | (0 / 0) | (0 / 0) |
| f3f32666cd0f38b0c9a1ef83f720 | repz repz es data16 int 0xf (6 / 28) | int 0xf (6 / 8) | int 0xf (6 / 7) | int 0x0F (6 / 8) |
| 2ef066f0480f3a2904b7 | cs lock data16 lock rex.W (bad) (8 / 33) | (0 / 0) | (0 / 0) | (0 / 0) |
| 67470f3875ae022de4a1517e90b4cb | addr32 rex.RXB (bad) (5 / 22) | (0 / 0) | (0 / 0) | (0 / 0) |
| 3664970f7a50db978a650a8288bee1 | ss fs xchg edi,eax (3 / 19) | xchg eax, edi (3 / 14) | xchg edi, eax (3 / 13) | xchg edi, eax (3 / 13) |
| f226f236458a49fb848d28 | repnz es repnz mov r9b,BYTE PTR ss:[r9-0x5] (8 / 44) | mov r9b, byte ptr ss:[r9 - 5] (8 / 30) | mov r9b, byte ptr [r9-0x5] (8 / 26) | mov r9b, ss:[r9-0x05] (8 / 21) |

# Code vs. Data

- **Some compilers** tightly interweave **data** (e.g., bytes, values) **within code**
  - Imprecision can create **trickle-down errors** in instruction recovery!
  - Example from OpenSSL (one of the most popular HTTPS libraries):

```
popfq           // original
.byte 0xf3,0xc3
.size AES_cbc_encrypt
.align 64
.LAES_Te
.long 0xa56363c6
```

```
popfq     // disassembled
repz retq
nop
nop
(bad)
movslq -0x5b(%rbx),%esp
```

# Questions?

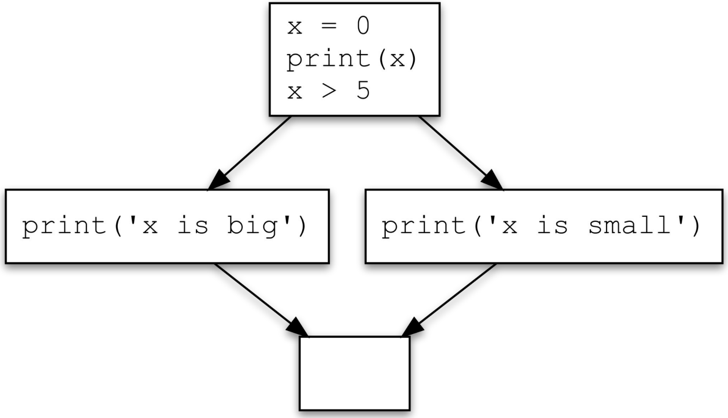
# Pillars of RE:
# Control Flow Recovery

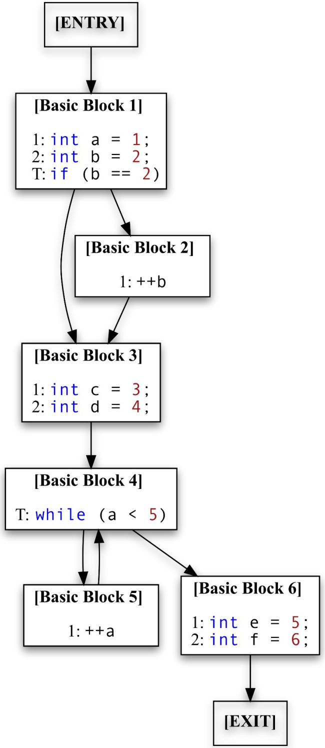
# Control Flow

- **What is it?**
  - **???**

# Control Flow

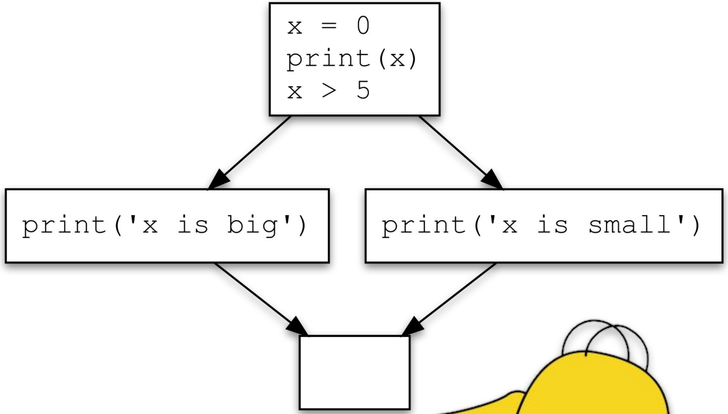
- **What is it?**
  - How execution flows from one application component to others
- **Why do we care?**
  - ???

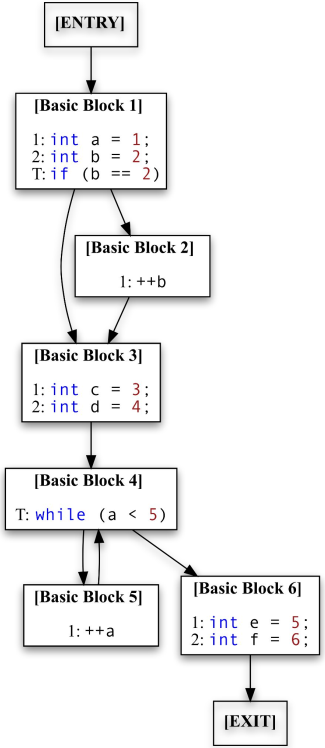


# Control Flow

- **What is it?**
  - How execution flows from one application component to others
- **Why do we care?**
  - Want to understand the entire program!



# Recovering Control Flow

- **Direct Edges**
  - Jump/call a function

```
jmp 0x4001AB3
```

Target is pre-set **statically**

# Recovering Control Flow

- **Direct Edges**
  - Jump/call a function
- **Indirect Edges**
  - Transfer to a register
  - Function pointers
  - Switch-case tables

`jmp 0x4001AB3`

Target is pre-set **statically**

`call %eax;` where?

Target found at **runtime**

# Recovering Control Flow

- **Direct Edges**
  - Jump/call a function
- **Indirect Edges**
  - Transfer to a register
  - Function pointers
  - Switch-case tables
- **“Pseudo” Edges**
  - Post-call returns

`jmp 0x4001AB3` — Target is pre-set **statically**

`call %eax;` where? — Target found at **runtime**

`ret;` goes where? — Necessary to recover **all paths**

# Recovering Control Flow

- **Direct Edges**
  - Jump/call a function
- **Indirect Edges**
  - Transfer to a register
  - Function pointers
  - Switch-case tables
- **“Pseudo” Edges**
  - Post-call returns
- **Tail Calls**
  - Call at function’s end

| | |
|---|---|
| `jmp 0x4001AB3` | Target is pre-set **statically** |
| `call %eax;` where? | Target found at **runtime** |
| `ret;` goes where? | Necessary to recover **all paths** |
| `jmp &foo;` call? | Expressed as **jumps**, not calls |

# Symbol Stripping

- **Debugging symbols:** maps instructions in the compiled binary program to their corresponding **variable**, **function**, or **line** in the source code.

```
int addition(int num1, int num2){
     return num1+num2;
}

int main(){
     int var1, var2;
     printf("Enter number 1: ");
     scanf("%d",&var1);
     printf("Enter number 2: ");
     scanf("%d",&var2);
     int res = addition(var1, var2);
     printf ("Output: %d", res);
     return 0;
}
```

# Symbol Stripping

- **Debugging symbols:** maps instructions in the compiled binary program to their corresponding **variable**, **function**, or **line** in the source code.
  - Makes **RE easy** if you have symbols…

```c
int addition(int num1, int num2){
    return num1+num2;
}

int main(){
    int var1, var2;
    printf("Enter number 1: ");
    scanf("%d",&var1);
    printf("Enter number 2: ");
    scanf("%d",&var2);
    int res = addition(var1, var2);
    printf ("Output: %d", res);
    return 0;
}
```

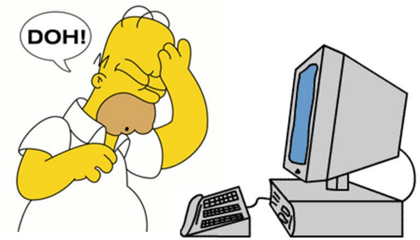
```
$ objdump --syms example | grep .text
0000000000001090 l F .text 0000000000000000  deregister_tm_clones
00000000000010c0 l F .text 0000000000000000  register_tm_clones
0000000000001100 l F .text 0000000000000000  __do_global_dtors_aux
0000000000001140 l F .text 0000000000000000  frame_dummy
0000000000001150 g F .text 0000000000000018  addition
0000000000001060 g F .text 0000000000000026  _start
0000000000001170 g F .text 0000000000000085  main
```

# Symbol Stripping

- **Debugging symbols:** maps instructions in the compiled binary program to their corresponding **variable**, **function**, or **line** in the source code.
  - Makes **RE easy** if you have symbols… but **often stripped** from the binary!

```c
int addition(int num1, int num2){
     return num1+num2;
}

int main(){
     int var1, var2;
     printf("Enter number 1: ");
     scanf("%d",&var1);
     printf("Enter number 2: ");
     scanf("%d",&var2);
     int res = addition(var1, var2);
     printf ("Output: %d", res);
     return 0;
}
```

```
$ objdump --syms example

example:     file format elf64-x86-64

SYMBOL TABLE:
no symbols
```



# Obfuscation

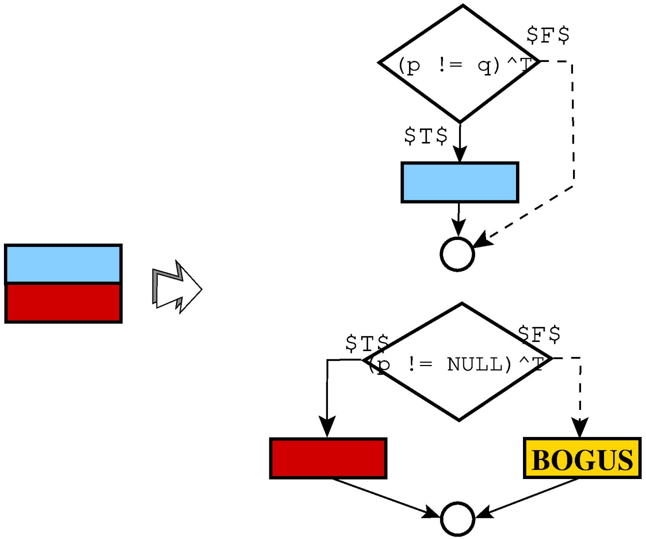
- **Obfuscation:** techniques designed to make third-party analysis **difficult**
  - **???**

# Obfuscation

- **Obfuscation:** techniques designed to make third-party analysis **difficult**
  - Developers want to keep their intellectual property secret to just themselves!

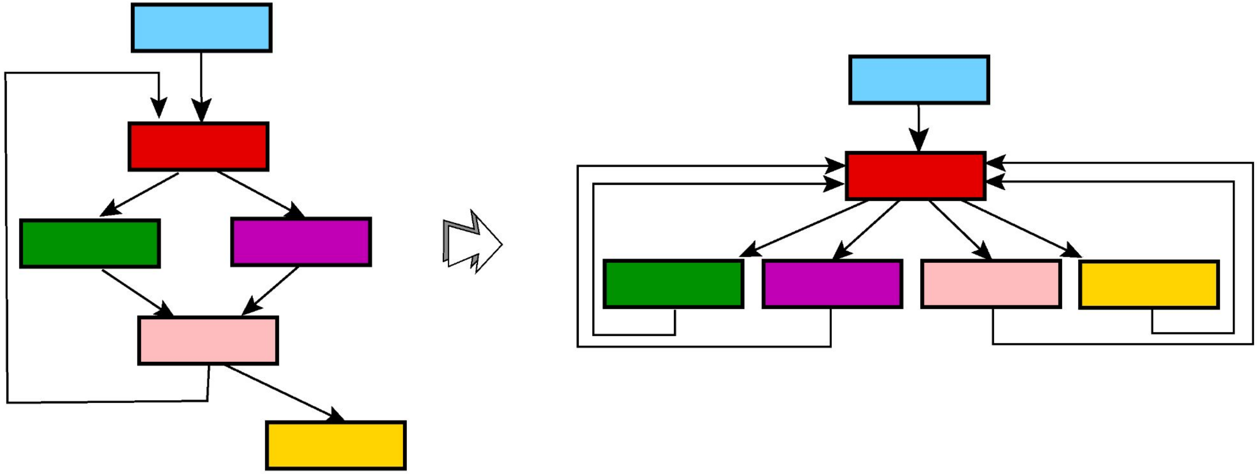
# Obfuscation

- **Obfuscation:** techniques designed to make third-party analysis **difficult**
  - Developers want to keep their intellectual property secret to just themselves!
  - **Example:** **opaque predicates** → introduces "fake" control-flow that is confusing!



# Obfuscation

- **Obfuscation:** techniques designed to make third-party analysis **difficult**
  - Developers want to keep their intellectual property secret to just themselves!
  - **Example:** **control-flow flattening** → removes any recognizable flow ordering



# Questions?

# Pillars of RE:
# Structure Recovery

# Program Structure

- **Why do we care?**
  - **???**

# Program Structure

- **Why do we care?**
  - Know how the code's parts work together

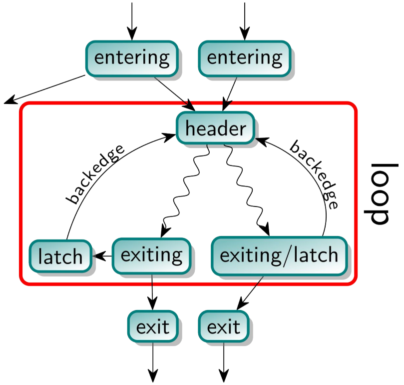
```
dec edi
push byte +0x1e
mov bh,0xb5
or al,0x12
push byte +0x4
jmp dword 0xb12b:0x7ba58ea0
mov cl,0x14
```

# Program Structure

- **Why do we care?**
  - Know how the code's parts work together

- **Examples:**
  - Basic Blocks
  - Loop Types
  - Recursion
  - Jump Tables
  - Functions

```
dec edi
push byte +0x1e
mov bh,0xb5
or al,0x12
push byte +0x4
jmp dword 0xb12b:0x7ba58ea0
mov cl,0x14
```

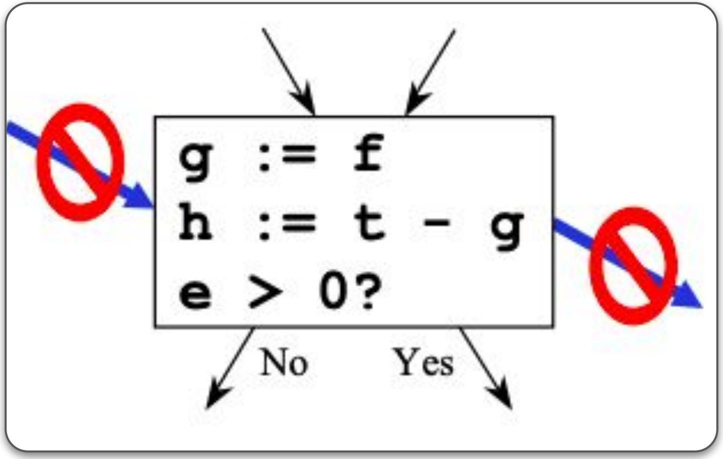


# Structure Recovery

- Largely **heuristic**-based
  - Construct-specific rules

# Structure Recovery

- Largely **heuristic**-based
  - Construct-specific rules
- **Basic Blocks:**
  - **Start:**
    - Target of a `jmp`
    - Target of a `call`
    - Target of a `ret`
  - **End:**
    - Ends in a `jmp`
    - Ends in a `call`
    - Ends in a `ret`



# Structure Recovery

- Largely **heuristic**-based
  - Construct-specific rules

- **Functions:**
  - **Start:**
    - Target of a `call`
    - Target of a tail call
    - A known prologue
    - A dispatch table entry
  - **End:**
    - Location of a `ret`
    - Location of a tail call
    - A known epilogue

```
push ebp
mov ebp, esp
sub esp, N
```

Prologue

```
mov esp, ebp
pop ebp
ret
```

Epilogue

```
switch(choice) {
    case 0 :
        result = add(first, second);
        break;
    case 1 :
        result = sub(first, second);
        break;
    case 2 :
        result = mult(first, second);
        break;
    case 3 :
        result = divide(first, second);
        break;
}
```

C-level Switch Table

# Questions?

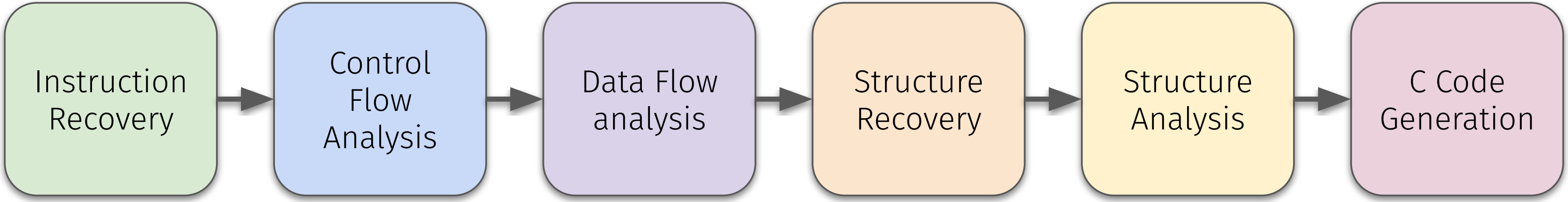
# RE Tasks: Decompilation

# Decompilation

- **Goal: ???**

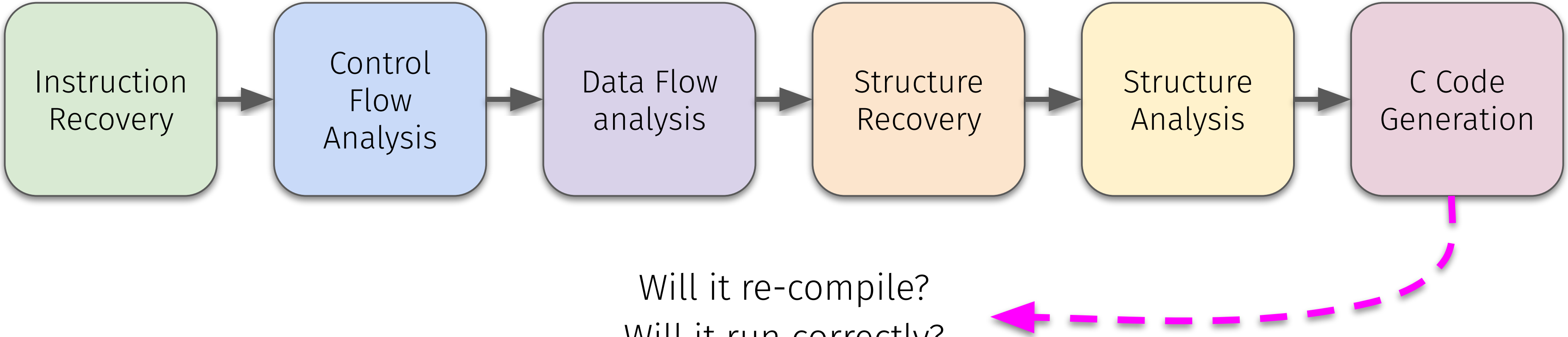
# Decompilation

- **Goal:** obtain **semantically-equivalent** source code from a compiled binary



# Decompilation

- **Goal:** obtain **semantically-equivalent** source code from a compiled binary
  - **In practice: really difficult** with little guarantee of success (**compilable** or **correct** code)




Will it re-compile?
Will it run correctly?
Is it human readable?

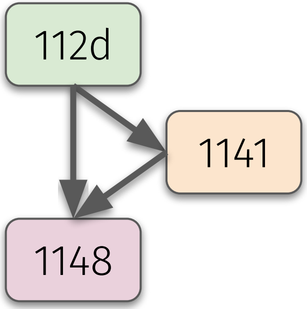
# Try it yourself!

```
112d: push %ebp
112e: mov %esp,%ebp    // mov src,dst
1131: mov %edi,$0x14(%ebp)
1134: mov $0x0,$0x4(%ebp)
113b: cmp $0x1,$0x14(%ebp)
113f: jne 1148
1141: add 0x1337,$0x4(%ebp)
1148: mov $0x0,%eax
114d: pop %ebp
114e: ret
114f: nop
```

https://rev.fish/files/bar2022_keynote.pdf

# Try it yourself!

```
112d: push %ebp
112e: mov %esp,%ebp    // mov src,dst
1131: mov %edi,$0x14(%ebp)
1134: mov $0x0,$0x4(%ebp)
113b: cmp $0x1,$0x14(%ebp)
113f: jne 1148
1141: add 0x1337,$0x4(%ebp)
1148: mov $0x0,%eax
114d: pop %ebp
114e: ret
114f: nop
```




Variables:

```
ebp-0x4:     foo
ebp-0x14:    bar
```

```
foo = 0; // 1134
if (bar == 1) { // 113b
    // 1141
    foo = foo + 0x1337;
}
return 0; // 1148
```

https://rev.fish/files/bar2022_keynote.pdf

# Popular Decompilers

- **Many decompilers** available today (both commercial and open-source)
  - Can lift binaries to **different languages** (e.g., C/C++, LLVM IR, custom IRs, etc.)

angr

IDA Pro

Binary Ninja

Ghidra

# Different Decompilers = Different Outputs

- **Example:** `HelloWorld` (ARM version) on DogBolt.org

**angr**

9.2.38

```c
int _init()
{
    unsigned int v0;  // [bp-0x8]
    unsigned int v1;  // [bp-0x4]
    unsigned int v2;  // lr
    unsigned int v3;  // r3

    v1 = v2;
    v0 = v3;
    return call_weak_fn();
}

int _start(unsigned int a0)
{
    unsigned int v0;  // [bp-0x8]
    unsigned int v1;  // [bp-0x4]
    unsigned int v2;  // [bp+0x0]

    v2 = stack_base + 4;
    v1 = a0;
    v0 = 0;
    __libc_start_main(); /* do not return */
}

int sub_10390()
{
    abort(); /* do not return */
}

int call_weak_fn()
{
    return;
    __gmon_start__();
    return;
}

int deregister_tm_clones()
```

**BinaryNinja**

3.3.3996 (e34a955e)

```c
int32_t _init(int32_t arg1, int32_t arg2)
{
    return call_weak_fn(arg1, arg2);
}

int32_t sub_10308()
{
    /* jump -> 0 */
}

void __libc_start_main(
    int32_t (* main)(int32_t argc, char** argv, char
    char** ubp_av, void (* init)(), void (* fini)(),
    void* stack_end) __noreturn
{
    /* tailcall */
    return __libc_start_main(main, argc, ubp_av, ini
}

int32_t getchar()
{
    /* tailcall */
    return getchar();
}

int32_t puts(char const* str)
{
    /* tailcall */
    return puts(str);
}

int32_t __gmon_start__()
{
    /* tailcall */
    return __gmon_start__();
}

```

**Ghidra**

10.2.2 (9813cde2)

```c
#include "out.h"



int _init(EVP_PKEY_CTX *ctx)

{
  int iVar1;

  iVar1 = call_weak_fn();
  return iVar1;
}



void __libc_start_main(void)

{
  __libc_start_main();
  return;
}



// WARNING: Unknown calling convention -- yet parame

int getchar(void)

{
  int iVar1;

  iVar1 = getchar();
  return iVar1;
}


```

**Hex-Rays**

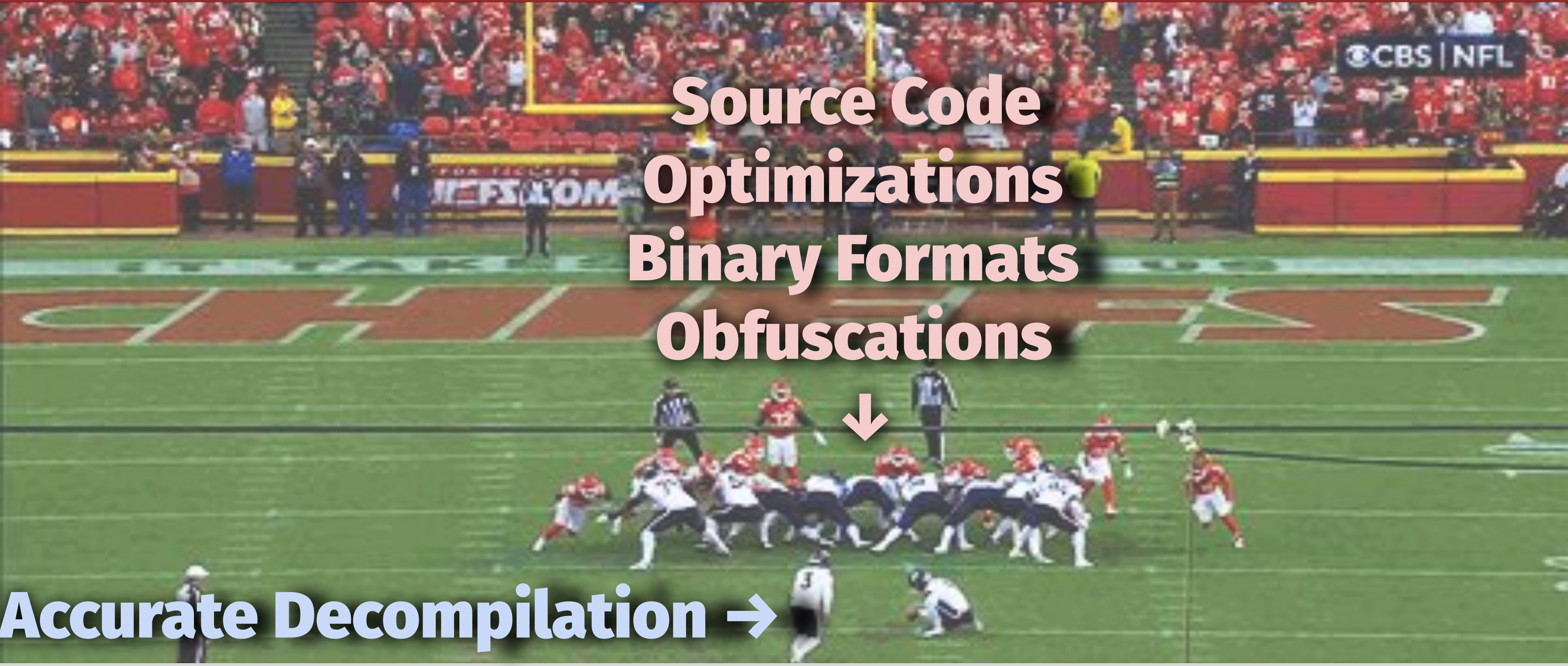
8.2.0.221215

```c
/* This file was generated by the Hex-Rays decompile
   Copyright (c) 2007-2021 Hex-Rays <info@hex-rays.c

   Detected compiler: GNU C++
*/

#include <defs.h>

#include <stdarg.h>


//-------------------------------------------------
// Function declarations

int init_proc();
void sub_10308();
// int __fastcall _libc_start_main(int (__fastcall *
// int getchar(void);
// int puts(const char *s);
// int _gmon_start__(void); weak
// void abort(void);
void __noreturn start(void (*a1)(void), int a2, int
int call_weak_fn();
char *deregister_tm_clones();
__int64 register_tm_clones();
char *_do_global_dtors_aux();
int __cdecl main(int argc, const char **argv, const
void term_proc();

//-------------------------------------------------
// Data declarations

char _bss_start; // weak
// extern _UNKNOWN __gmon_start__; weak


//----- (000102FC) --------------------------------
```

# Questions?

# Our Research:
# Fuzzing Decompilers' Correctness

# Challenges to Binary Decompilation




Source Code
Optimizations
Binary Formats
Obfuscations
↓

Accurate Decompilation →

# How can we test decompilers?

- **Differential testing:**
  - Mutate source code
  - Decompile, then recompile
  - Compare programs' output
  - **Non-equivalence** = **bug**

Globals & Values

```
long g0     = 0;
long *g1[1] = {&g0};
int  g2     = 1;
int  *g3    = &g2;
```

Mutated Source

```
int main () {
  if (!g0) {
    ++g2;
    g3 = (int *)1;
    if (g1[0] != (long *)1) {
      g2 = 7UL
        % (g1[0] != (long *)1 ?
        1 : g1[0] == (long *)1);
    } else {
      g2 = 0;
    }
  }
}
```

**Compare:** `g0, g1, g2, g3`

# How can we test decompilers?

- **Differential testing:**
  - Mutate source code
  - Decompile, then recompile
  - Compare programs' output
  - **Non-equivalence** = **bug**
- ... but is that it?
  - Compilers?
  - Formats?
  - Optimizations?

```c
long g0     = 0;
long *g1[1] = {&g0};
int  g2     = 1;
int  *g3    = &g2;
```

Globals & Values

```c
int main () {
  if (!g0) {
    ++g2;
    g3 = (int *)1;
    if (g1[0] != (long *)1) {
      g2 = 7UL
        % (g1[0] != (long *)1 ?
        1 : g1[0] == (long *)1);
    } else {
      g2 = 0;
    }
  }
}
```

Mutated Source

**Compare:** g0, g1, g2, g3

# How can we test decompilers?

- **Differential testing:**
  - Mutate source code
  - Decompile, then recompile

- ...
  - Formats?
  - Optimizations?

```
long g0      = 0;
long *g1[1] = {&g0};
int  g2      = 1;
int  *g3     = &g2;
```

Globals & Values

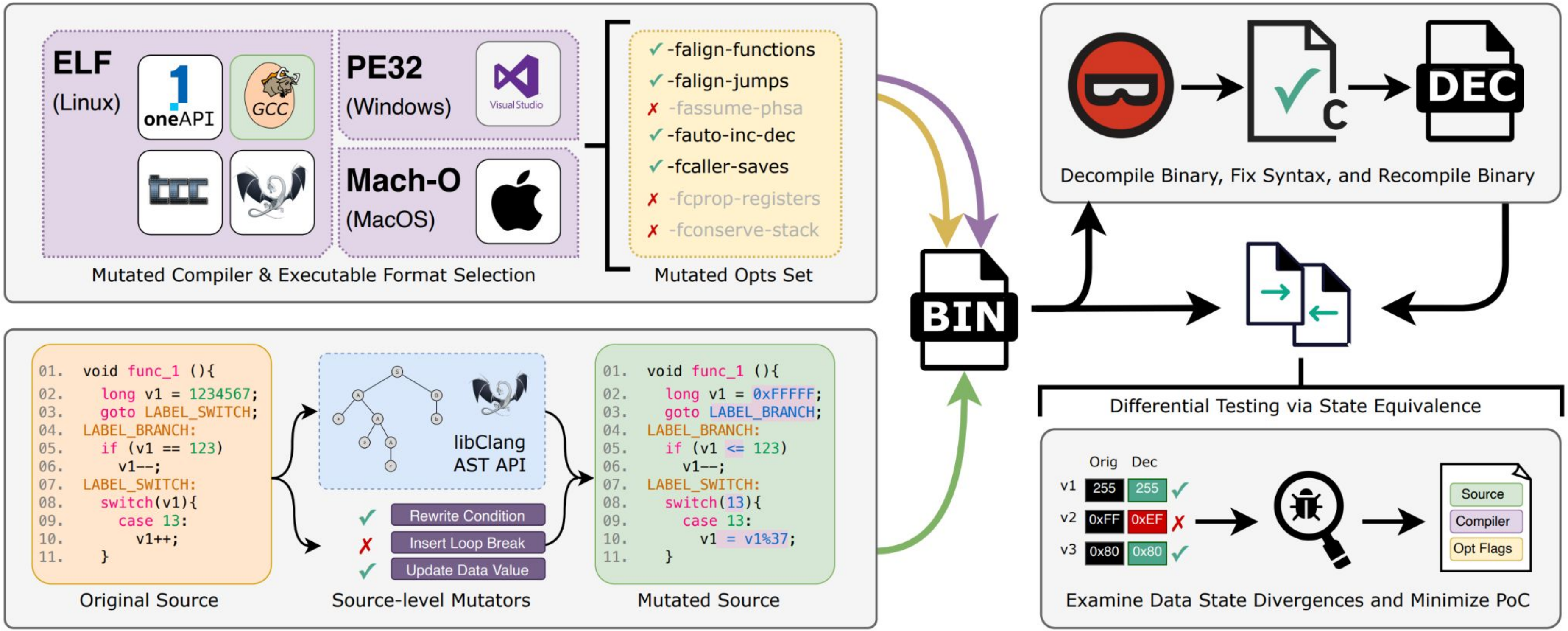
```
      } else {
        g2 = 0;
      }
    }
}
```

...utated Source

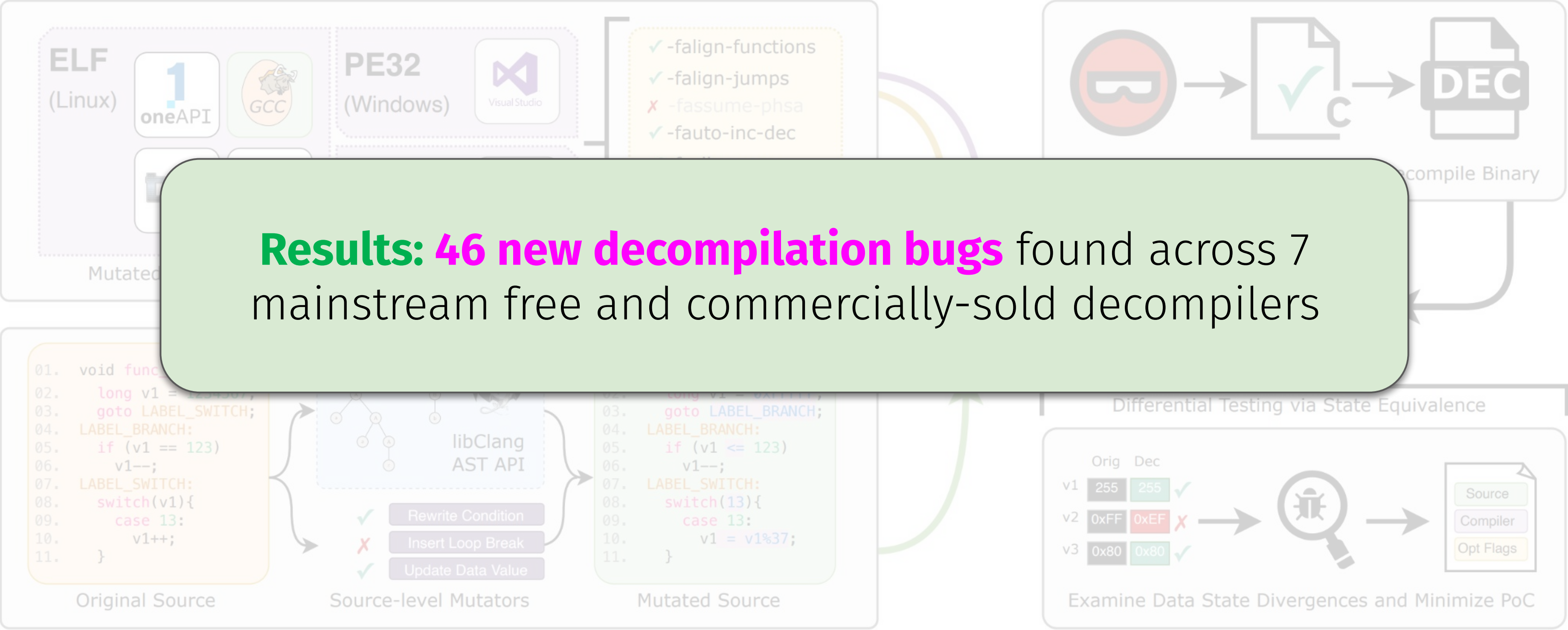
**Compare:** g0, g1, g2, g3

**Challenge:** How to **systematically explore *all* factors**—and combinations thereof—**uniquely influencing binary code**?

# Our Approach: Mutate *Everything*



# Our Approach: Mutate *Everything*




**Results: 46 new decompilation bugs** found across 7 mainstream free and commercially-sold decompilers

# Some Interesting Decompilation Bugs We've Found

- **#1:** Erroneous recovery of **floating-point data** as **integers**
  - Affects Angr, Binary Ninja, and Reko

```
movsd xmm0,[rbp];
double v0 = 1.0;
```

(a) Original

```
int64_t v1 =
0x3FF0000000000000;
```

(b) Decompiled

# Some Interesting Decompilation Bugs We've Found

- **#2:** Mis-handling of **Mach-O** (MacOS) and **ELF** (Linux) **calling conventions**
  - Affects Reko

```
mov r8,  z
mov rdx, y
mov rcx, x
func(x, y, z);
```

(a) PE convention & code.

```
mov rdi, x
mov rsi, y
mov rdx, z
func(z, y, x);
```

(b) Mach-O & ELF decompiled.

# Some Interesting Decompilation Bugs We've Found

- **#3:** Erroneous recovery of **switch-case** case logic
  - Affects Binary Ninja

- **Result:** completely different execution paths at runtime!
  - Deemed a high-severity bug
  - Since found other instances

```
int var = 0;
switch(var){
  case 0:      Reached?
    var = 5;     ✔
    break;
  case 1:
    ...
  case 2:
    ...
  default:
    idx = 0;     ✗
    break;
}
```

(a) Original

```
int var = 0;
if (var == 2)
  ...
}
else{
  if (var == 0)  Reached?
    var = 5;        ✔
  if (var == 1)
    ...
  if (var > 2){
    idx = 0;        ✔
  }
}
```

(b) Decompiled

# Some Interesting Decompilation Bugs We've Found

## BINARY NINJA BLOG

### RESTRUCTURING THE BINARY NINJA DECOMPILER

Rusty Wagner 2024-06-19 reversing, decompiler

For the upcoming Binary Ninja 4.1, we will be releasing a new implementation of our decompiler's control flow recovery. You can try it today by switching to the development channel and updating to the latest build. It isn't fully optimized yet and may produce non-optimal results in some cases, but there should already be improvements in readability of the output, including a reduction in nesting depth and a significant reduction in the complexity of conditional expressions.

This new implementation aims to improve the readability of the decompiler output while simultaneously improving accuracy. It also aims to significantly improve maintainability, allowing us to iterate on our decompiler faster. We have additionally added a new suite of tests to allow us to make changes to the decompiler and have more confidence that the changes haven't caused regressions in accuracy.

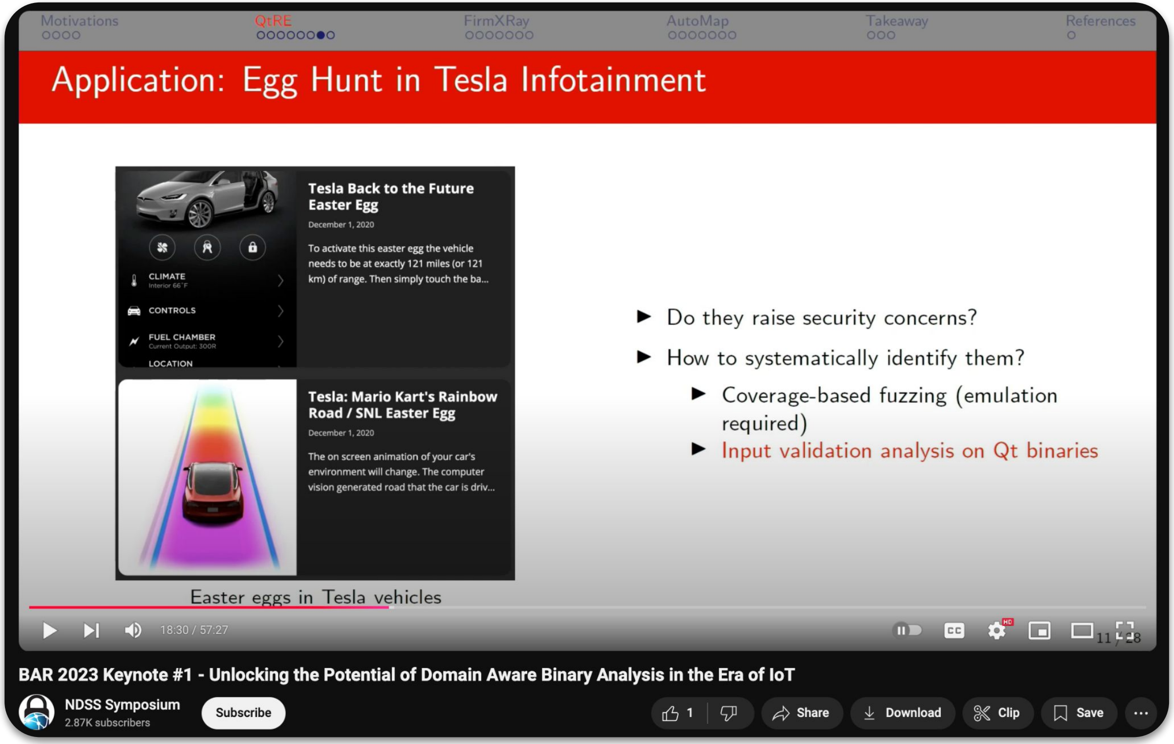
### FOOTNOTES / CREDITS

The following resources may be helpful for understanding additional approaches to decompilation, provided motivating examples, or directly inspired work described here. Additionally, one of the primary motivations for this improvement was a privately reported decompilation flaw from Zao Yang and Dr. Stefan Nagy of the FuTURES³ Lab. Keep an eye on their forthcoming research and we're grateful for their notification!

https://binary.ninja/2024/06/19/restructuring-the-decompiler.html

# Supplemental Content: Domain-specific RE

- Dr. Zhiqiang Lin's keynote at BAR'23
- **Lots of cool bugs!**
  - Tesla Infotainment
  - "Super Apps"
  - And more!
- **Check it out!**



# Next time on CS 4440…

Today's Security Ecosystem
Bug Bounties, CTF Competitions
Career Paths in Cyber Security