Week 12: Lecture B
Software Reverse Engineering

Thursday, November 16, 2023
Announcements

- **Project 3** grades are now available on **Canvas**

- **Statistics:**
  - Average score: **90%**

- **Fantastic job!**
Announcements

- **Project 3** grades are now available on **Canvas**

- Think we made an error? Request a regrade!
  - Valid regrade requests:
    - You have verified your solution is correct (i.e., we made an error in grading)

**Project 3 Regrade Requests** (see **Piazza** pinned link): Submit by **11:59 PM** on **Monday 11/20** via **Google Form**
Announcements

- **Project 4: NetSec** released
  - **Deadline:** Thursday, December 7th by 11:59PM

---

**Project 4: Network Security**

*Deadline: Thursday, December 7 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.
<table>
<thead>
<tr>
<th>Task</th>
<th>Progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working on Part 1.0: Password Cracking</td>
<td>0%</td>
</tr>
<tr>
<td>Working on Part 1.1: Port Scanning</td>
<td>0%</td>
</tr>
<tr>
<td>Working on Part 1.2: Anomalous Activity</td>
<td>0%</td>
</tr>
<tr>
<td>Finished Part 1, working on Part 2!</td>
<td>0%</td>
</tr>
<tr>
<td>None of the above</td>
<td>0%</td>
</tr>
</tbody>
</table>
Participation

- **Preliminary Participation** score on Canvas
  - Designed to help you gauge where you’re at
  - Not your final participation score!

- Cliff notes:
  - Everyone receives full Piazza participation
  - Remaining 50% of score based on lectures

- Total of **22 lectures** (11 weeks * 2 lectures)
  - Participation capped at 70% (~15 / 22) of lectures
Preliminary Participation score on Canvas
- Designed to help you gauge where you’re at
- Not your final participation score!

Cliff notes:
- Everyone receives full Piazza participation
- Remaining 50% of score based on lectures

Total of 22 lectures (11 weeks * 2 lectures)
- Participation capped at 70% (~15 / 22) of lectures

Score lower than expected?
Check that you have a PollEverywhere account
(or just insert your UID in your account name)

Haven’t been attending?
There’s still time to get the points!
No Class Next Week
Announcements

See Discord for meeting info!

www.utahsec.com
Questions?
Last time on CS 4440...

Adversarial Machine Learning
Data Poisoning Attacks
Evasion Attacks
ML Ethics
Uses of ML

- **ML is all around us!**
  - By now, we interact with ML-based systems daily
  - Expanding every year

- **Real-world examples:**
  - Marketing optimization
  - Fraud detection
  - Stock trading
  - Self-guided cars
  - Cancer screening
  - Others?
ML in a Nutshell

- ML aims to produce a model that correctly makes inferences about inputs
  - E.g., category of a given input fruit (apple or orange)
  - E.g., likelihood of a person being a reliable creditor
**Training vs. Testing**

- **Training Data:**
  - ???
Training vs. Testing

- **Training Data:**
  - The data used to **build** the model
  - May be labeled or unlabeled
  - **Example:** pictures of all orange types

- **Testing Data:**
  - ???
Training vs. Testing

- **Training Data:**
  - The data used to **build** the model
  - May be labeled or unlabeled
  - **Example:** pictures of all orange types

- **Testing Data:**
  - Data used to test model’s **correctness**
  - Should be **separate** from training data!
  - **Example:** tangerines, clementines, kiwis
Threat Model

- Data poisoning (causative attack): 
  - ???
Data poisoning (causative attack):
- Attackers perturb training set to fool the model (insert/modify data, alter data labels)
- Attackers attempt to influence or corrupt the ML model or the ML algorithm itself

Malicious Training Data

Model: wrong result
Threat Model

- **Evasion attack** (exploratory attack):
  - ???
**Threat Model**

- **Evasion attack** (exploratory attack):
  - Attackers don’t tamper with ML model, but instead cause it to produce adversarial outputs.
  - Attack on the model’s testing phase—by far the most common attack type seen today.

**Diagram:***
- **Attacker:** crafted input
- **Model:** wrong result
- **Refine malicious input**
# Adversarial Examples

<table>
<thead>
<tr>
<th>Distance/Angle</th>
<th>Subtle Poster</th>
<th>Subtle Poster Right Turn</th>
<th>Camouflage Graffiti</th>
<th>Camouflage Art (LISA-CNN)</th>
<th>Camouflage Art (GTSRB-CNN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5' 0°</td>
<td><img src="image1" alt="Stop Sign" /></td>
<td><img src="image2" alt="Right Turn Sign" /></td>
<td><img src="image3" alt="Graffiti" /></td>
<td><img src="image4" alt="Art (LISA-CNN)" /></td>
<td><img src="image5" alt="Art (GTSRB-CNN)" /></td>
</tr>
<tr>
<td>5' 15°</td>
<td><img src="image1" alt="Stop Sign" /></td>
<td><img src="image2" alt="Right Turn Sign" /></td>
<td><img src="image3" alt="Graffiti" /></td>
<td><img src="image4" alt="Art (LISA-CNN)" /></td>
<td><img src="image5" alt="Art (GTSRB-CNN)" /></td>
</tr>
<tr>
<td>10' 0°</td>
<td><img src="image1" alt="Stop Sign" /></td>
<td><img src="image2" alt="Right Turn Sign" /></td>
<td><img src="image3" alt="Graffiti" /></td>
<td><img src="image4" alt="Art (LISA-CNN)" /></td>
<td><img src="image5" alt="Art (GTSRB-CNN)" /></td>
</tr>
<tr>
<td>10' 30°</td>
<td><img src="image1" alt="Stop Sign" /></td>
<td><img src="image2" alt="Right Turn Sign" /></td>
<td><img src="image3" alt="Graffiti" /></td>
<td><img src="image4" alt="Art (LISA-CNN)" /></td>
<td><img src="image5" alt="Art (GTSRB-CNN)" /></td>
</tr>
<tr>
<td>40' 0°</td>
<td><img src="image1" alt="Stop Sign" /></td>
<td><img src="image2" alt="Right Turn Sign" /></td>
<td><img src="image3" alt="Graffiti" /></td>
<td><img src="image4" alt="Art (LISA-CNN)" /></td>
<td><img src="image5" alt="Art (GTSRB-CNN)" /></td>
</tr>
</tbody>
</table>

| Targeted-Attack Success | 100% | 73.33% | 66.67% | 100% | 80% |

Stefan Nagy
Adversarial Examples

Figure 1: We create an adversarial patch that is successfully able to hide persons from a person detector. Left: The person without a patch is successfully detected. Right: The person holding the patch is ignored.
Adversarial Examples

Deep Fake via **Generative Adversarial Network (GAN)**
Adversarial Examples

Deep Fake via Generative Adversarial Network (GAN)
Ethical Considerations
Questions?
This time on CS 4440...

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

Source Code

Preprocessor
- substitutes #include directives with content of included files

Compiler
- generates binary machine code

Linker
- combines binary machine code and connects function calls

Binary Executable

### Source Code

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World\n";
}
```

```cpp
... cout = ...
...
int main() {
    std::cout << "Hello World\n";
}
```
How Software is Built

Source Code

standard library header file

iostream

... 
... cout = ... 
...

hello.cpp

#include <iostream>

int main() {
    std::cout <<
    "Hello World\n";
}

Preprocessor

substitutes #include directives with content of included files

Compiler

generates binary machine code

Linker

combines binary machine code and connects function calls

Binary Executable

hello.o

binary object file

hello.exe
How Software is Built

- **Source Code**
  - Standard library header file
  - Source code files (e.g., `iostream`, `hello.cpp`)
  - Example code:
    ```cpp
    #include <iostream>
    int main() {
      std::cout << "Hello World\n";
    }
    ```

- **Preprocessor**
  - Substitutes `#include` directives with content of included files

- **Compiler**
  - Generates binary machine code

- **Linker**
  - Combines binary machine code and connects function calls

- **Binary Executable**
  - Compiled program file (e.g., `hello.exe`)
How Software is Built

Source Code

Preprocessor
substitutes #include directives with content of included files

Compiler
generates binary machine code

Linker
combines binary machine code and connects function calls

Binary Executable

std::cout << "Hello World\n";

int main() {

... cout = ...
...

int main() {

std::cout << "Hello World\n";
}
How Software is Built

Source Code

Preprocessor

substitutes #include directives with content of included files

Compiler

generates binary machine code

Linker

combines binary machine code and connects function calls

Binary Executable

iostream

... 
... cout = ... 
...

hello.cpp

#include <iostream>

int main() {
  std::cout <<
  "Hello World\n";
}

... 
... cout = ... 
...

int main() {
  std::cout <<
  "Hello World\n";
}

hello.o

binary object file

hello.exe
How Software is Built

Source Code

Preprocessor
substitutes #include directives with content of included files

Compiler
generates binary machine code

Linker
combines binary machine code and connects function calls

Binary Executable

Today’s Focus

#include <iostream>

int main()
{
    std::cout << "Hello World\n";
}

int main()
{
    std::cout << "Hello World\n";
}
Closed-source Software

- It’s everywhere!
Closed-source Software

- It’s everywhere!
  - **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
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!

Stefan Nagy
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**
  - ???

---

```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;
}
```
RE Tasks

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

- **Decompilation**
  - Machine code to human readable source code

- **Rewriting**
  - Add more functionality and rebuild executable

```
000000003 4F     dec edi
000000004 6A1E   push byte +0x1e
000000006 B7B5   mov bh,0xb5
000000008 0C12   or al,0x12
00000000A 6A04   push byte +0x4
00000000C EAA08EA57B2BB1 jmp dword 0xb12b:0x7ba58ea0
000000013 B114   mov cl,0x14
```

```c
int main(void)
{
    char local_54 [64];
    int modified;
    modified = 0;
    if (modified == 0) {
        puts("Try again?");
    } else {
        puts("You have changed the 'modified' variable");
    }
    return 0;
}
```
<table>
<thead>
<tr>
<th>Experience with RE?</th>
<th>0%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disassembly (e.g., objdump)</td>
<td>0%</td>
</tr>
<tr>
<td>Decompilation (e.g., Ghidra, IDA)</td>
<td>0%</td>
</tr>
<tr>
<td>Something else!</td>
<td>0%</td>
</tr>
<tr>
<td>None of the above (totally fine!)</td>
<td>0%</td>
</tr>
</tbody>
</table>
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?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPS</td>
<td>Maximum of Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MAXSD</td>
<td>Return Maximum Scalar Double-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MAXSS</td>
<td>Return Maximum Scalar Single-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MFENCE</td>
<td>Memory Fence</td>
</tr>
<tr>
<td>MINPD</td>
<td>Minimum of Packed Double-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MINPS</td>
<td>Minimum of Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MINSID</td>
<td>Return Minimum Scalar Double-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MINSS</td>
<td>Return Minimum Scalar Single-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MONITOR</td>
<td>Set Up Monitor Address</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
</tr>
<tr>
<td>MOV (1)</td>
<td>Move to/from Control Registers</td>
</tr>
<tr>
<td>MOV (2)</td>
<td>Move to/from Debug Registers</td>
</tr>
<tr>
<td>MOVAPD</td>
<td>Move Aligned Packed Double-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MOVAPS</td>
<td>Move Aligned Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MOVBE</td>
<td>Move Data After Swapping Bytes</td>
</tr>
<tr>
<td>MOVD</td>
<td>Move Doubleword/Move Quadword</td>
</tr>
<tr>
<td>MOVDDUP</td>
<td>Replicate Double FP Values</td>
</tr>
<tr>
<td>MOVDIR64B</td>
<td>Move 64 Bytes as Direct Store</td>
</tr>
<tr>
<td>MOVDIRI</td>
<td>Move Doubleword as Direct Store</td>
</tr>
<tr>
<td>MOVDDQ2Q</td>
<td>Move Quadword from XMM to MMX Technology Register</td>
</tr>
</tbody>
</table>
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 = ???**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPS</td>
<td>Maximum of Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MAXSD</td>
<td>Return Maximum Scalar Double-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MAXSS</td>
<td>Return Maximum Scalar Single-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MFENCE</td>
<td>Memory Fence</td>
</tr>
<tr>
<td>MINSQD</td>
<td>Minimum of Packed Double-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MINPS</td>
<td>Minimum of Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MINSQD</td>
<td>Return Minimum Scalar Double-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MINSS</td>
<td>Return Minimum Scalar Single-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MONITOR</td>
<td>Set Up Monitor Address</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
</tr>
<tr>
<td>MOV (1)</td>
<td>Move to/from Control Registers</td>
</tr>
<tr>
<td>MOV (2)</td>
<td>Move to/from Debug Registers</td>
</tr>
<tr>
<td>MOVAPD</td>
<td>Move Aligned Packed Double-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MOVAPS</td>
<td>Move Aligned Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MOVBE</td>
<td>Move Data After Swapping Bytes</td>
</tr>
<tr>
<td>MOVD</td>
<td>Move Doubleword/Move Quadword</td>
</tr>
<tr>
<td>MOVQDUP</td>
<td>Replicate Double FP Values</td>
</tr>
<tr>
<td>MOVDIR64B</td>
<td>Move 64 Bytes as Direct Store</td>
</tr>
<tr>
<td>MOVDIRI</td>
<td>Move Doubleword as Direct Store</td>
</tr>
<tr>
<td>MOVQDQ2Q</td>
<td>Move Quadword from XMM to MMX Technology Register</td>
</tr>
</tbody>
</table>
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

#### Table of Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPS</td>
<td>Maximum of Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MAXSD</td>
<td>Return Maximum Scalar Double-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MAXSS</td>
<td>Return Maximum Scalar Single-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MFENCE</td>
<td>Memory Fence</td>
</tr>
<tr>
<td>MINPD</td>
<td>Minimum of Packed Double-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MINPS</td>
<td>Minimum of Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MINSD</td>
<td>Return Minimum Scalar Double-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MINSS</td>
<td>Return Minimum Scalar Single-Precision Floating-Point Value</td>
</tr>
<tr>
<td>MONITOR</td>
<td>Set Up Monitor Address</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
</tr>
<tr>
<td>MOV (1)</td>
<td>Move to/from Control Registers</td>
</tr>
<tr>
<td>MOV (2)</td>
<td>Move to/from Debug Registers</td>
</tr>
<tr>
<td>MOVAPD</td>
<td>Move Aligned Packed Double-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MOVAPS</td>
<td>Move Aligned Packed Single-Precision Floating-Point Values</td>
</tr>
<tr>
<td>MOVBE</td>
<td>Move Data After Swapping Bytes</td>
</tr>
<tr>
<td>MOVD</td>
<td>Move Doubleword/Move Quadword</td>
</tr>
<tr>
<td>MOVDDUP</td>
<td>Replicate Double FP Values</td>
</tr>
<tr>
<td>MOVDIR64B</td>
<td>Move 64 Bytes as Direct Store</td>
</tr>
<tr>
<td>MOVDIRI</td>
<td>Move Doubleword as Direct Store</td>
</tr>
<tr>
<td>MOVQ2Q</td>
<td>Move Quadword from XMM to MMX Technology Register</td>
</tr>
</tbody>
</table>
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

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
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

- **Recursive Descent**
  - Follow all control-flow transfers
  - jmp 0x100 \(\rightarrow\) start decoding instructions at address 0x100
  - Stop when you’ve covered all possible control-flow paths

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

**Challenge:** data within code

**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 end of .TEXT section

- **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:** Compilers lay code sequentially for compactness
**Challenge:** Data within code

**Intuition:** Following 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!

<table>
<thead>
<tr>
<th>Byte offset</th>
<th>Legacy Prefix</th>
<th>Opcode Prefix</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

- Mod
  - R/M: optional
  - SIB: optional

- x86_32: optional

- x86_32: required

- Displacement

- Immediate

- Immediate
CISC Architectures

- Example of byte offsets and possible decodings:

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

```
js 0xffffffffffe840f58
```
CISC Architectures

- Example of byte offsets and possible decodings:

\[ \text{mov} \quad \text{BYTE PTR} \ [\text{rdx}+0xf], \text{dl} \]
\[ \text{test} \quad \text{ah}, \text{ch} \]
Example of byte offsets and possible decodings:

\begin{align*}
0x0F & \quad 0x88 \quad 0x52 \quad 0x0F \quad 0x84 \quad 0xEC \\
\text{add} & \quad \text{eax, 0x40080f20} \\
\text{in} & \quad \text{al, dx}
\end{align*}
## Instruction Decoder Bugs

### Results from Trail of Bits’ Mishegos fuzzer:

<table>
<thead>
<tr>
<th>input</th>
<th>worker</th>
<th>hex.worker</th>
<th>opatone</th>
<th>rox</th>
<th>hex.worker</th>
<th>neelf</th>
<th>rox</th>
<th>hex.worker</th>
<th>term</th>
<th>rox</th>
<th>hex.worker</th>
<th>tyulin</th>
<th>rox</th>
</tr>
</thead>
<tbody>
<tr>
<td>05672f029bda23654</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>2626276044d93817114ac5f9d</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>05662c2602e3e9059068d1f8c52</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>b7904e3d0f0f0f6646</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>05622c5202f5a710d0bc093023f3</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>06332c60f3d45909b5d15f5533e</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>06328c520ff3a710f53c0e590235d6</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>b62c3b005e36315c6</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>b62c3b005e36315c6</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>06332c60f3d45909b5d15f5533e</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>06328c520ff3a710f53c0e590235d6</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>06332c60f3d45909b5d15f5533e</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>06328c520ff3a710f53c0e590235d6</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>06332c60f3d45909b5d15f5533e</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>06328c520ff3a710f53c0e590235d6</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
<tr>
<td>06332c60f3d45909b5d15f5533e</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
<td>push (1/1)</td>
</tr>
</tbody>
</table>
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):

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

```assembly
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?**
  - ???

```
x = 0
print(x)
if x > 5:
    print('x is big')
    print('x is small')
else:
    print('x is small')
```

Diagram:

- **[ENTRY]**
- **[Basic Block 1]**
  1: int a = 1;
  2: int b = 2;
  3: if (b == 2):
- **[Basic Block 2]**
  1: ++b
- **[Basic Block 3]**
  1: int c = 3;
  2: int d = 4;
- **[Basic Block 4]**
  1: while (a < 5):
- **[Basic Block 5]**
  1: ++a
- **[Basic Block 6]**
  1: int e = 5;
  2: int f = 6;
- **[EXIT]**
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
  
  ```
  jmp 0x4001AB3
  ```

- **Indirect Edges**
  - Transfer to a register
  - Function pointers
  - Switch-case tables

  ```
  call %eax; where?
  ```

Target is pre-set **statically**

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
```

```
call %eax; where?
```

```
ret; goes where?
```

Target is pre-set **statically**

Target found at **runtime**

Necessary to recover **all paths**
Recovering Control Flow

- **Direct Edges**
  - Jump/call a function
  - Example: `jmp 0x4001AB3`
  - Target is pre-set *statically*

- **Indirect Edges**
  - Transfer to a register
  - Function pointers
  - Switch-case tables
  - Example: `call %eax; where?`
  - Target found at *runtime*

- **“Pseudo” Edges**
  - Post-call returns
  - Example: `ret; goes where?`
  - Necessary to recover *all paths*

- **Tail Calls**
  - Call at function’s end
  - Example: `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.

```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;
}
```
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;
}
```

```bash
$ 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**: 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?
  - ???

Stefan Nagy
Program Structure

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

```assembly
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

```plaintext
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

```c
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;
}
```

- **Prologue**
  - `push ebp`
  - `mov ebp, esp`
  - `sub esp, N`

- **Epilogue**
  - `mov esp, ebp`
  - `pop ebp`
  - `ret`
Questions?
RE Tasks: Decompilation
Decompilation

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

- In practice: *really difficult* with little guarantee of success (*compilable* or *correct* code)

---

Instruction Recovery ➔ Control Flow Analysis ➔ Data Flow analysis ➔ Structure Recovery ➔ Structure Analysis ➔ C Code Generation

Will it 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

Try it yourself!

```assembly
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

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

Decompile Tools

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

Example: HelloWorld (ARM version) on DogBolt.org
Questions?
My Prior Work: Static Binary Rewriting for Fuzzing
Fuzzing from Source vs. Binary Executables

**Compiler Instrumentation**

- Semantically rich

Low (18–32%) overhead
Enhanced via **code xform**
Fuzzing from Source vs. Binary Executables

**Compiler Instrumentation**
- semantically rich
- Low (18–32%) overhead
- Enhanced via code xform

**Binary-only Instrumentation**
- semantically opaque
- Up to 10,000% slower
- Outweighed by overhead

Enhanced via code xform
Fuzzing from Source vs. Binary Executables

**Motivation:** Can compiler instrumentation’s transformation and speed be extended to the fuzzing of binary-only targets?

Compiler Instrumentation vs. Binary-only Instrumentation

- **Compiler Instrumentation:**
  - Semantically rich
  - Low (18–32%) overhead
  - Enhanced via code xform

- **Binary-only Instrumentation:**
  - Semantically opaque
  - Up to 10,000% slower
  - Outweighed by overhead
Closing the Fuzzing Instrumentation Gap

What properties make compilers ideal for speed and code transformation?

Key insight: to achieve compiler-quality speed and transformation, must match how compilers handle these four key design elements:

- Invoked Inline
- Inserted Statically
- Register Liveness
- Broad Scalability
Z AFL: a Platform for Binary Fuzzing Instrumentation

- **Statically**-inserted, **inlined**, and register-optimized instrumentation
- Adapted from the Zipr binary rewriting project
- Support for **x86-64 ELF binaries** (and cross-platform support for **PE32+**)

Diagram:

1. **Static Rewriting Component**
   - Binary Rewriter
     - Build Binary Representation
     - IR Data Struct
     - Reconstitute Output Binary
   - Original Binary
   - Modified IR
   - Output Binary

2. **Z AFL Transform & Inst. Phases**
   - P1: Control-Flow Options
     - Specify Optimizations
     - Optimized Control-flow Graph
   - P2: Control-Flow Analysis
     - Specify Analyses
     - Extract Meta-characteristics
   - P3: Inst. Point Selection
     - Meta-characteristic Data
     - Location Selection
   - P4: Inst. Application
     - Selection, Liveness, Inst. Templates
     - Apply Instrumentation
Implement a suite of five **compiler-based code enhancements** used in fuzzing

**Performance-enhancing:**
- Single-successor path pruning
- Dominator tree CFG pruning
- Instrumentation downgrading

**Feedback-enhancing:**
- Sub-instruction profiling
- Context sensitivity tracking

ZAFL’s **low-level API** brings a **semantic richness** to the otherwise **semantically-opaque** environment of binary fuzzing
Evaluation: ZAFL’s Scalability
Evaluation: ZAFL’s Scalability

Evaluate ZAFL on 56 executables (33 open- and 23 closed-source)

- Linux and Windows binaries
- PIE and non-PIE code layouts
- File size 100KB–100MB
- 100–1,000,000 basic blocks
Evaluation: Fuzzing Closed-source Binaries

- By carefully matching compilers’ key attributes, ZAFL introduces compiler-quality speed and transformation capabilities to binary-only fuzzing:

**Scalability:** supports many characteristics/sizes/complexity/platforms

**Performance:** within 10% of compiler-based fuzzing instrumentation

**Bug discovery:** 26–131% more bugs than leading binary fuzzing instrumenters

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Location</th>
<th>AFL-Dyninst</th>
<th>AFL-QEMU</th>
<th>ZAFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>heap overflow</td>
<td>nconvert</td>
<td>Cannot find</td>
<td>18.3 hrs</td>
<td>12.7 hrs</td>
</tr>
<tr>
<td>stack overflow</td>
<td>unrar</td>
<td>Cannot find</td>
<td>12.3 hrs</td>
<td>9.04 hrs</td>
</tr>
<tr>
<td>heap overflow</td>
<td>pngout</td>
<td>12.6 hrs</td>
<td>6.26 hrs</td>
<td>1.93 hrs</td>
</tr>
<tr>
<td>use-after-free</td>
<td>pngout</td>
<td>9.35 hrs</td>
<td>4.67 hrs</td>
<td>1.44 hrs</td>
</tr>
<tr>
<td>heap overread</td>
<td>libida64.so</td>
<td>23.7 hrs</td>
<td>Cannot find</td>
<td>2.30 hrs</td>
</tr>
</tbody>
</table>

**ZAFL’s Mean Relative Decrease**

-660% -113%

![Graph showing unrar](image-url)
Questions?
Next time on CS 4440...

Side Channels and Hardware Security