Abstraction Recovery for Scalable Static Binary Analysis

Edward J. Schwartz
Software Engineering Institute
Carnegie Mellon University
The Gap Between Binary and Source Code

push %ebp
mov %esp,%ebp
sub $0x10,%esp
movl $0x1,-0x4(%ebp)
jmp 1d <f+0x1d>
mov -0x4(%ebp),%eax
imul 0x8(%ebp),%eax
mov %eax,-0x4(%ebp)
subl $0x1,0x8(%ebp)
cmpl $0x1,0x8(%ebp)
jg f <f+0xf>
mov -0x4(%ebp),%eax
leave
ret

int f(int c) {
    int accum = 1;
    for (; c > 1; c--) {
        accum = accum * c;
    }
    return accum;
}
Static Binary Analysis

Automatic extraction of facts about binary programs without executing them.
Static Binary Analysis Strengths

• **High Coverage**
  – Reason about most or all possible executions

• **Safe**
  – Does not execute (possibly unsafe) code

• **Widely Applicable**
  – Source code not needed
  – Useful for end-users, researchers, sysadmins
Primary Challenge: Scalability

Largest Program

Static Binary Code Analysis Tools

Static Source Code Analysis Tools

Largest Program
The Gap Between Binary and Source Code

```
push    %ebp
mov     %esp,%ebp
sub     $0x10,%esp
movl    $0x1,-0x4(%ebp)
jmp     1d <f+0x1d>
mov     -0x4(%ebp),%eax
imul    0x8(%ebp),%eax
mov     %eax,-0x4(%ebp)
subl    $0x1,0x8(%ebp)
cmpl    $0x1,0x8(%ebp)
jg      f <f+0xf>
mov     -0x4(%ebp),%eax
leave   
ret
```

```
int f(int c) {
    int accum = 1;
    for (; c > 1; c--) {
        accum = accum * c;
    }
    return accum;
}
```
1. Choose abstractions

2. Recover abstractions

3. Scalable, high-level reasoning
“Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction.”

Chikofsky and Cross
Software Reuse and Reverse Engineering in Practice
Reverse Engineering

Abstraction → More Abstract → Less Detail → Binary

More Abstract
Less Detail
Abstraction Recovery

1. Choose abstractions
2. Recover abstractions
3. Scalable, high-level reasoning

Compilied C Programs

Decompiation

Reverse Engineering

- Functions
  - Types
  - Variables

Static Source Code Analysis
Outline

• Introduction

• Recovering Abstractions
  – C abstractions (Phoenix Decompiler)
  – Gadget abstractions (Q ROP Compiler)

• Future Work and Conclusions
int f (int x) {
    int y = 1;
    while (x > y) {
        y++;
    }
    return y;
}

int f (int a) {
    int v = 1;
    while (a > v++)
    {};
    return v;
}
The Phoenix Decompiler

- Designed for abstraction recovery
  - Correctness (new)
    - Prior work: focus on manual reverse engineering
  - Effective abstraction recovery

- Design: series of stages
  - Each stage recovers a different abstraction
  - Some are new; some are not
Phoenix Overview

CFG Recovery → Variable and Type Recovery

Source-code Output or Analysis → Control Flow Structuring

New in Phoenix

int f (int x) {
    int y = 1;
    while (x > y) {
        y++;
    }
    return y;
}
Control Flow Structuring

```java
if (e) {
    ...
} else {
    ...
}
```

Control Flow Structuring

```java
if (e) {
    ...
} else {
    ...
}
```

Compilation
Structural Analysis

- Iteratively match patterns to CFG
  - Collapse matching regions

If-then

While

Sequence
...;
while (...) { if (...) {...} else {...} }; 
...; ...;
Structural Analysis Property Checklist

1. Correctness
Structural Analysis Property Checklist

1. Correctness
   - Not originally intended for decompilation
   - Structure can be incorrect for decompilation
Semantics Preservation

• Reductions preserve meaning of program
Structural Analysis Property Checklist

1. Correctness
   - Not originally intended for decompilation
   - Structure can be incorrect for decompilation

2. Effective abstraction recovery
Structural Analysis Property Checklist

1. Correctness
   - Not originally intended for decompilation
   - Structure can be incorrect for decompilation

2. Effective abstraction recovery
   - Graceless failures for unstructured programs
     • break, continue, and gotos
     • Failures cascade to large subgraphs
This break edge prevents progress.
Iterative Refinement

- Remove edges that are preventing a match
  - Represent in decompiled source as `break`, `goto`, `continue`
  - Run on remaining graph

Allows structuring algorithm to make more progress
Iterative Refinement

Original

```java
s1;
while (e1) {
    if (e2) { break; }
    s2;
}
s3;
```

Decompiled

```java
s1;
while (e1) {
    if (e2) { break; }
    s2;
}
s3;
```
Large Scale Experiment Details

• How does Phoenix compare with state of the art?

• Measure impact of:
  – Semantics preservation
  – Iterative refinement

• Other decompilers
  – Hex-Rays (industry state of the art)
  – Boomerang (academic state of the art)
Large Scale Experiment Details

• How does Phoenix compare with state of the art?

• Measure impact of:
  – Semantics preservation
  – Iterative refinement

• Other decompilers
  – Hex-Rays (industry state of the art)
  – Boomerang (academic state of the art)
    • Did not terminate in <1 hour for most programs
Large Scale Experiment Details

• GNU coreutils 8.17, compiled with gcc
  – Programs of varying complexity
  – Test suite

• Metrics
  – Correctness
    • Number of decompiled utilities that pass unit tests
    • Has not been done before on large scale!
  – Control-flow structure recovery
    • Count number of goto statements
Number of Correct Utilities

Phoenix: 60
Hex-Rays: 28
Number of Correct Utilities

Phoenix: 60

Phoenix (w/o semantics preservation): 46
Number of Correct Utilities

- **All Utilities**: 107
- **Hex-Rays**: 28
- **Phoenix**: 60
- **Phoenix (w/o semantics preservation)**: 46
Correctness

• Any incorrect abstraction can cause incorrect decompilation
  – Hex Rays
    • ?
  – Phoenix
    • All (known) correctness errors attributed to type recovery
      – Undiscovered variables
    • No known problems in control flow structuring
Control Flow Structure: Gotos Emitted (Fewer is Better)

- Phoenix: 40
- Hex-Rays: 51
Control Flow Structure: Gotos Emitted (Fewer is Better)

Phoenix: 40
Phoenix (w/o iterative refinement): 1229
Hex-Rays: 51
Outline

• Introduction

• Recovering Abstractions
  – C abstractions (Phoenix Decompiler)
  – Gadget abstractions (Q ROP Compiler)

• Future Work and Conclusions
OS Defenses

• All major operating systems employ defenses
  – DEP: Data Execution Prevention
  – ASLR: Address Space Layout Randomization

• Make reliable exploitation difficult
  – How difficult?
Simple Control-Flow Hijack Exploit

Exploit

Shellcode  Padding  Pointer

Computation  Control
Data Execution Prevention (DEP)

Exploit

Shellcode  Padding  Pointer

Crash

User input is non-executable and cannot be writable
Bypassing DEP

• **Goal**: Specify exploit computation even when DEP is enabled

• **Return-oriented Programming** [Shacham 2007]
  – Use existing instructions from program it to create self-contained gadgets
  – Chain gadgets together to encode computation
Example: How can we write to memory without shellcode?
Return-oriented Programming

Exploit

nextaddr
addr3
address
addr2
value
stack

Gadgets

addr1
pop %eax
ret

daddr2
pop %ebx
ret

daddr3
movl %eax, (%ebx)
ret
Gadgets as Abstractions

• Gadgets are behavior specifications
  – Load constant
  – Store to memory
  – Don’t need to reason about low-level behavior to combine them
Address Space Layout Randomization (ASLR)

ASLR disabled

Exploit

Gadgets

ASLR enabled

Gadgets

Exploit

Crash

ASLR: Addresses are unpredictable
Return-oriented Programming + ASLR

• Randomized code can’t be used for ROP

• But ASLR implementations do not randomize all code...
(Typical) Randomized Code in Linux

Unrandomized

Randomized

Libc

Stack

Heap

Executable

PPREW: Abstraction Recovery
12/9/2014
© 2014 Carnegie Mellon University
Modern Exploitation using ROP

• Program image is often the only unrandomized code
  – Small
  – Program-specific

• How much unrandomized code does an attacker need to use ROP?

We need a graduate student with a lot of free time
We need automatic ROP techniques that can work with the program image
Q: Automatic ROP System
Q: ROP Overview

Source P → Discovery → Assignment → Computation → Arrangement → Discovery

Abstraction Recovery
High-level Reasoning
Gadget Discovery

- **Discovery**: Does instruction sequence do something we can use for our computation?
- Fast randomized test for every program location (thousands or millions)

```assembly
sbb %eax, %eax;
neg %eax; ret
```
Randomized Testing

Before

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0x0298a7bc</td>
</tr>
<tr>
<td>CF</td>
<td>0x1</td>
</tr>
<tr>
<td>ESP</td>
<td>0x81e4f104</td>
</tr>
</tbody>
</table>

Move Code:

```asm
sbb %eax, %eax;
neg %eax; ret
```

After

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0x1</td>
</tr>
<tr>
<td>ESP</td>
<td>0x81e4f108</td>
</tr>
<tr>
<td>EBX</td>
<td>0x0298a7bc</td>
</tr>
</tbody>
</table>

OutReg <- InReg

Semantic Definition For Move
## Q’s Semantic Definitions/ Gadget Types

<table>
<thead>
<tr>
<th>Gadget Type</th>
<th>Semantic Definition</th>
<th>Real World Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoveRegG</td>
<td>Out &lt;- In</td>
<td>xchg %eax, %ebp; ret</td>
</tr>
<tr>
<td>LoadConstG</td>
<td>Out &lt;- Constant</td>
<td>pop %ebp; ret</td>
</tr>
<tr>
<td>ArithmeticG</td>
<td>Out &lt;- In1 + In2</td>
<td>add %edx, %eax; ret</td>
</tr>
<tr>
<td>LoadMemG</td>
<td>Out &lt;- M[Addr + Offset]</td>
<td>movl 0x60(%eax), %eax; ret</td>
</tr>
<tr>
<td>StoreMemG</td>
<td>M[Addr + Offset] &lt;- In</td>
<td>mov %dl, 0x13(%eax); ret</td>
</tr>
<tr>
<td>ArithmeticLoadG</td>
<td>Out +&lt;- M[Addr + Offset]</td>
<td>add 0x1376dbe4(%ebx), %ecx; (...) ; ret</td>
</tr>
<tr>
<td>ArithmeticStoreG</td>
<td>M[Addr + Offset] +&lt;- In</td>
<td>add %al, 0x5de474c0(%ebp); ret</td>
</tr>
</tbody>
</table>
Randomized Testing

• Randomized testing quickly rules out non-gadgets
  – Fast
  – Enables more expensive second stage

• Second stage: program verification
Connection to Program Verification

sum = 0
while (n > 0) {
    sum += n;
    n--;
}

sbb %eax, %eax
neg %eax; ret

EAX <- CF

Does the post-condition always hold after executing program?
Gadget Verification

sbb %eax, %eax
neg %eax; ret
EAX <- CF

Weakest Precondition

θ

Validity Check

Valid (Gadget)

Invalid (not Gadget)
# Semantic-based Gadget Discovery

- Q is better at finding gadgets than I am!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imul $1, %eax, %ebx ret</td>
<td>Move %eax to %ebx</td>
</tr>
<tr>
<td>lea (%ebx,%ecx,1), %eax ret</td>
<td>Store %ebx+%ecx in %eax</td>
</tr>
<tr>
<td>sbb %eax, %eax; neg %eax ret</td>
<td>Move carry flag to %eax</td>
</tr>
</tbody>
</table>
Q: ROP Overview

Source P → Discovery → Assignment → Arrangement → Computation

Abstraction Recovery

High-level Reasoning
Research Questions

How much unrandomized code is sufficient to create ROP payloads?

– Detail: payloads call any functions in libc
  – system, execv, connect, mprotect
ROP Success Probability

Call libc functions in 80% of programs $\geq$ true (20KB)
Can Q automatically add ROP payloads to existing exploits for real programs?
Real Exploits

- Q was able to automatically add ROP to nine exploits downloaded from exploit-db.com

<table>
<thead>
<tr>
<th>Name</th>
<th>Total Time</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free CD to MP3 Converter</td>
<td>130s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>Fatplayer</td>
<td>133s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>A-PDF Converter</td>
<td>378s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>A-PDF Converter (SEH exploit)</td>
<td>357s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>MP3 CD Converter Pro</td>
<td>158s</td>
<td>Windows 7</td>
</tr>
<tr>
<td>rsync</td>
<td>65s</td>
<td>Linux</td>
</tr>
<tr>
<td>opendchub</td>
<td>225s</td>
<td>Linux</td>
</tr>
<tr>
<td>gv</td>
<td>237s</td>
<td>Linux</td>
</tr>
<tr>
<td>Proftpd</td>
<td>44s</td>
<td>Linux</td>
</tr>
</tbody>
</table>
Demo!
Outline

• Introduction

• Recovering Abstractions
  – C abstractions (Phoenix Decompiler)
  – Gadget abstractions (Q ROP Compiler)

• Future Work and Conclusions
Abstraction Recovery Questions

- **Systems**: How do we build systems that
  - Recover abstractions?
  - Use abstractions?

- **Theory**: When is it possible to recover abstractions?
  - Observable behaviors preserved by compilation

- **Scalability**: How does recovering and utilizing abstractions improve scalability?
  - ROP (150x)
  - C verification (15x)
Future Work

• Certified decompilation
  – Prove that binary $\rightarrow$ C translation is correct

• Optimal abstraction recovery
  – Provably optimal algorithms (i.e., minimum gotos)

• Additional abstractions & architectures
  – C++, ARM, Dalvik
Thanks to My Great Co-authors

Thanassis Avgerinos
David Brumley
JongHyup Lee
Maverick Woo
Conclusion

• Abstraction Recovery
  – Recovering abstractions helps static binary analysis

• Phoenix decompiler
  – Goal: Correct, effective decompilation
  – New control-flow structuring algorithm

• Q ROP Compiler
  – Takeaway: Unrandomized code is dangerous
  – 20KB makes DEP+ASLR ineffective
Thanks 😊

• Questions?

Edward J. Schwartz
eschwartz@cert.org
http://www.ece.cmu.edu/~ejschwar