Introduction to Binary Analysis and Binary Rewriting

Zhiqiang Lin

Department of Computer Science and Engineering
The Ohio State University

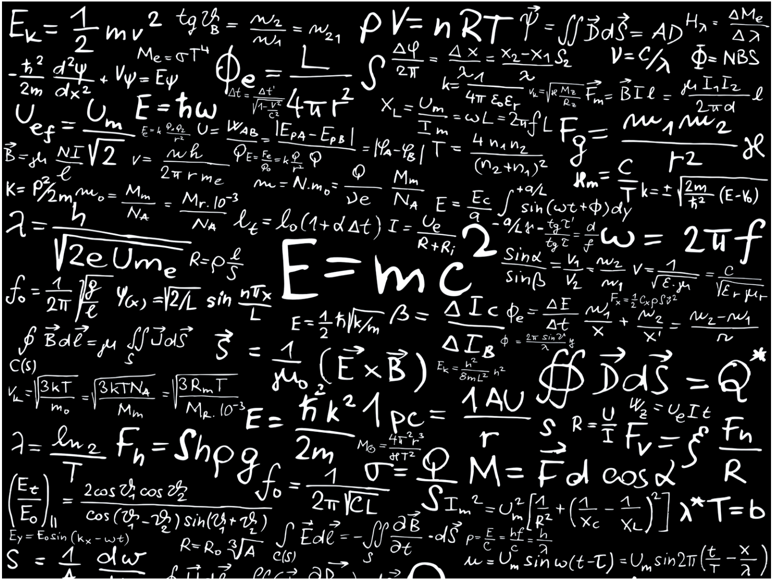
7/19/2018

Outline

Outline

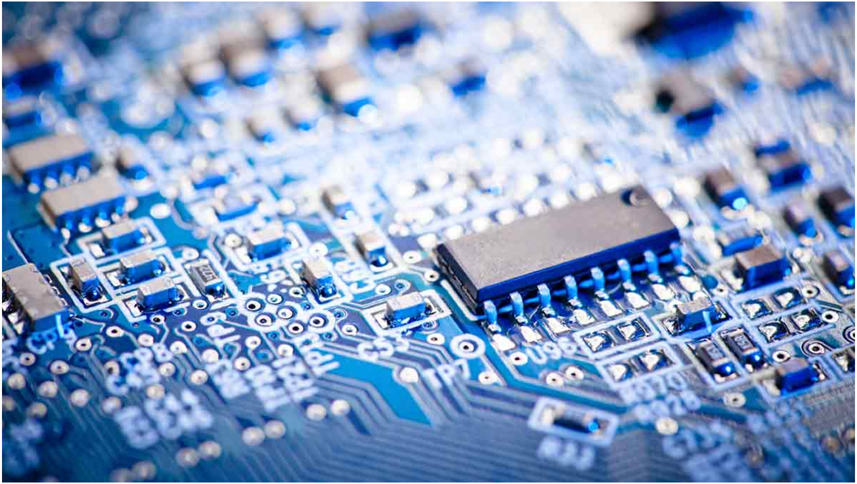
What's Running in Our Cyberspace

What's Running in Our Cyberspace



What's Running in Our Cyberspace

What's Running in Our Cyberspace



What's Running in Our Cyberspace

What is Binary Analysis

The process of (automatically) reasoning/deriving properties about the structure/behavior/syntactics/semantics/anything of your interest of binary programs

```
zlin@zlin-desktop:~/$ hexdump -C /bin/ls|less
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  a4 45 40 00 00 00 00 00  |..>......E@.....|
00000020  40 00 00 00 00 00 00 00  70 96 01 00 00 00 00 00  |@.......p.......|
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1c 00 1b 00  |....@.8...@.....|
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00  |@.@.....@.@.....|
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 00 00  |................|
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00  |................|
00000080  38 02 00 00 00 00 00 00  38 02 40 00 00 00 00 00  |8.......8.@.....|
00000090  38 02 40 00 00 00 00 00  1c 00 00 00 00 00 00 00  |8.@.............|
...
```

Why Binary Code?

Access to the source code often is not possible:

- Proprietary software packages. (Volks Wagon's cheating software)
- Stripped executables.
- Proprietary libraries
- Malicious software (exploits), e.g., stuxnet

Why Binary Code?

Access to the source code often is not possible:

- Proprietary software packages. (Volks Wagon's cheating software)
- Stripped executables.
- Proprietary libraries
- Malicious software (exploits), e.g., stuxnet

Binary code is the only authoritative version of the program.

- Binary code is everywhere
- Changes occurring in the compile, optimize and link steps can create non-trivial semantic differences from the source and binary.
- What you see is not what you execute (WYSINWYX problem)

Why Binary Code?

- Windows
  - Login process keeps a user's password in the heap after a successful login
- To minimize data lifetime
  - clear buffer
  - call free()

```
memset(buffer, '\0', len);
free(buffer);
```

Why Binary Code?

- Windows
  - Login process keeps a user's password in the heap after a successful login
- To minimize data lifetime
  - clear buffer
  - call free()

- But . . .
  - the compiler might optimize away the buffer-clearing code ("useless-code" elimination)

```
memset(buffer, '\0', len);
free(buffer);
```

⇨

```
free(buffer);
```

Why Binary Code: Backdoor

Linux embedded device: HTTP server for management and video monitoring, with a known backdoor.

Backdoor!!!

- ➔ Username: 3sadmin
- ➔ Password: 27988303

```
LDR     R1, =a3sadmin ; "3sadmin"
MOV     R0, R7  ; s1
BL      strcmp
CMP     R0, #0
LDR     R1, =a27988303 ; "27988303"
MOV     R0, R4  ; s1
```

Heffner, Craig. "Finding and Reversing Backdoors in Consumer Firmware." EELive! (2014).

What to Reason About in Binary Code?

The process of (automatically) reasoning/deriving properties about the structure/behavior/syntactics/semantics/anything of your interest of binary programs

1. What are the program's variables and their types?
2. What are the program's parameters and their types?
3. Where could this indirect jump go?
4. What function could be called at this indirect call site?
5. What could this dereference operation access/affect?
6. What kind of object is allocated at this allocation site?
7. What could the value held in V eventually affect?
8. What could have affected the value of V?
9. What are the statements (at instruction level) that contribute to the execution of i?
10. ...

Outline

Challenges: Abstraction Recovery

The first step in any binary code analysis is to reconstruct the abstractions distilled after compilation, such as recognizing the instructions, operand, opcode, variables, basic blocks, and control flows

Challenges: Abstraction Recovery

The first step in any binary code analysis is to reconstruct the abstractions distilled after compilation, such as recognizing the instructions, operand, opcode, variables, basic blocks, and control flows

Challenges

- Code/Data distinction
- Variable x86 instruction size
- Indirect Branches
- Functions without explicit CALL
- Position independent code (PIC)
- ...

It will be easier to recover these abstractions by using **dynamic analysis**, but will be much more challenge in **static analysis**.

Challenges: Path Coverage, and Path Explosion

For both static analysis and dynamic analysis, how to model the program execution path (too conservative, or too simple), and how to trigger the program path (especially for dynamic analysis) is another key challenge

Challenges: Path Coverage, and Path Explosion

For both static analysis and dynamic analysis, how to model the program execution path (too conservative, or too simple), and how to trigger the program path (especially for dynamic analysis) is another key challenge

Static analysis

- Too conservative
- Too many paths
- Impossible path

Dynamic analysis

- A single path
- Cover more path
- Test case generation

A Surprise:

Many source-level issues gone

1. Use of multiple source languages
2. In-lined assembly code
3. Avoid building problems
4. Analyze the actual libraries
5. ...

Many people inspecting binaries

1. IDA Pro Users
2. Anti-malware companies
3. Computer Emergency Response Teams
4. Malware writers
5. ...

A Surprise:

- Many exploits utilize platform-specific quirks
  - non-obvious and unexpected
  - compiler artifacts (choices made by the compiler)
    - Memory layout
      — padding between fields of a struct
      — which variables are adjacent
    - register usage
    - execution order
    - optimizations performed
    - compiler bugs

Outline

Basic Approaches to disassemble code

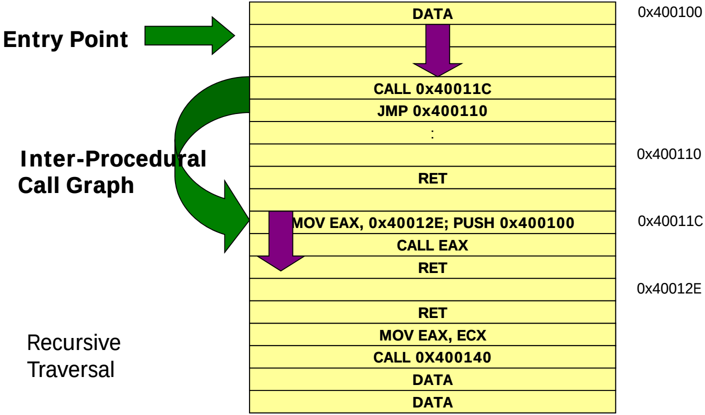
Linear sweep algorithm

- Start with program entry point, proceed to disassemble instructions sequentially
- Key assumption: all instructions appear one after the next, without any gaps
  - Violated in most code (presence of data or padding)

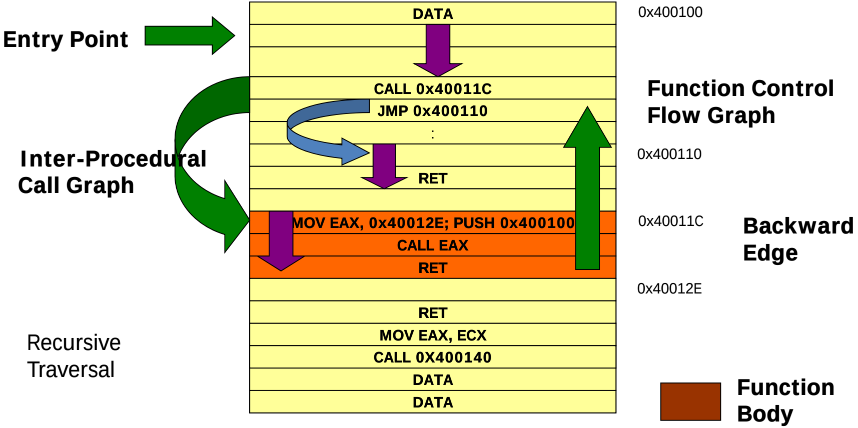
Recursive Traversal Algorithm

- After a control-flow transfer instruction (CTI), proceed to disassemble target address
- For conditional CTI and non-CTI, proceed to disassemble next instruction
- Key problems
  - Code reached only through indirect CTIs
  - Functions that do not return in the usual way

Examples

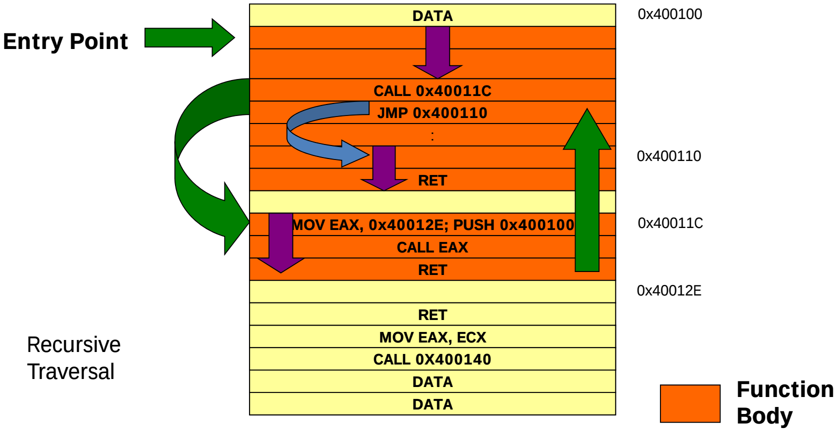

Entry Point
Inter-Procedural Call Graph
Recursive Traversal
DATA
CALL 0x40011C
JMP 0x400110
:
RET
MOV EAX, 0x40012E; PUSH 0x400100
CALL EAX
RET
RET
MOV EAX, ECX
CALL 0X400140
DATA
DATA
0x400100
0x400110
0x40011C
0x40012E

Examples



Entry Point
Inter-Procedural Call Graph
Recursive Traversal
DATA
CALL 0x40011C
JMP 0x400110
:
RET
MOV EAX, 0x40012E; PUSH 0x400100
CALL EAX
RET
RET
MOV EAX, ECX
CALL 0X400140
DATA
DATA
0x400100
0x400110
0x40011C
0x40012E
Function Control Flow Graph
Backward Edge
Function Body

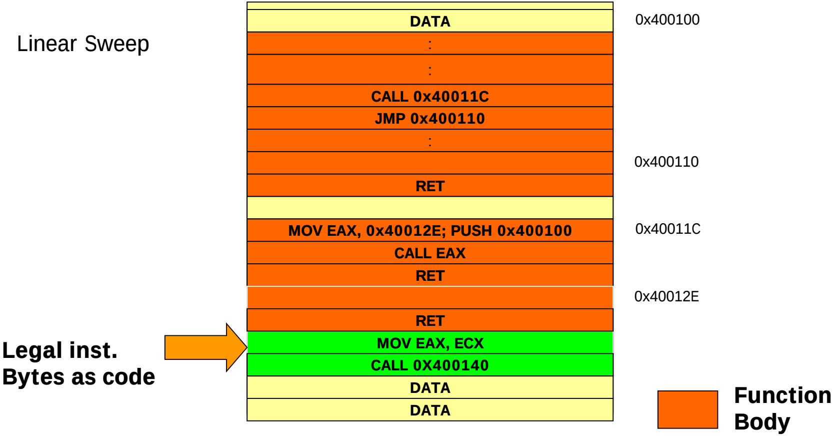
Examples



Entry Point
DATA
CALL 0x40011C
JMP 0x400110
:
RET
MOV EAX, 0x40012E; PUSH 0x400100
CALL EAX
RET
RET
MOV EAX, ECX
CALL 0X400140
DATA
DATA
0x400100
0x400110
0x40011C
0x40012E
Recursive Traversal
Function Body

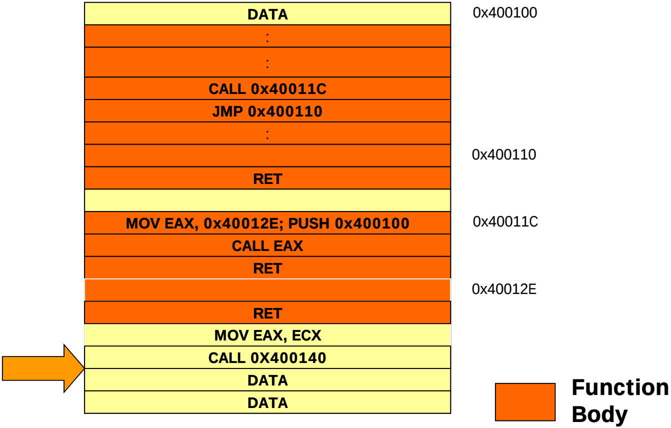
Examples

Linear Sweep



Examples

Error Recovery Heuristics



Code sequence not ending with JMP/RET → not code

Disassemble package

1. Udis86 is an easy-to-use, minimalistic disassembler library (libudis86) for the x86 class of instruction set architectures http://udis86.sourceforge.net/.
2. libdasm, a disassembly library. https://code.google.com/p/libdasm/.
3. pydasm https://github.com/OpenRCE/pydbg
4. Capstone is a lightweight multi-platform, multi-architecture disassembly framework. http://www.capstone-engine.org/
5. ...

Basic Techniques

1. Data Flow Analysis
   - Data dependency
   - Taint analysis
   - Point-to analysis
2. Control Flow Analysis
   - Control flow graph
   - Call graph
   - Control dependency
3. Program Slicing
4. Symbolic Execution

Static Analysis, Dynamic Analysis, Symbolic Execution

BAP BAT CodeReason

radare2

vivisect Hex-Ray IDA

rdis Valgrind

amoco fuzzgrind gdb

SemTrax

angr

BitBlaze

JARVIS

BARF

Jakstab

klee/s2e

insight

PIN QEMU

Bindead

Triton

PySysEmu TEMU PEMU

miasm CodeSurfer

paimei

Common Tools

1. Static analysis
   - IDA Pro, BinNav
   - BAP
2. Dynamic analysis
   - PIN
   - QEMU
   - PEMU
3. Symbolic execution
   - FuzzBall, Fuzzgrind
   - S2E
   - Angr

Outline

Applications of Binary Analysis in Security

Use Cases

1. Reverse engineering (knowing the secrets)
2. Vulnerability discovery/fuzzing
3. Exploit generation
4. Software verification
5. Program testing
6. Binary hardening
7. ...

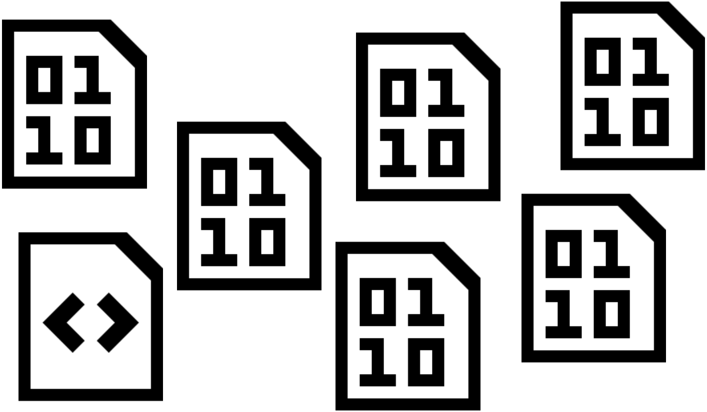
Applications: Vulnerability Discovery

Vulnerability Discovery

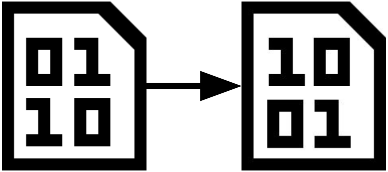
Basic Approaches

1. Static Analysis
   - "You can't " / "You might be able to"
   - Based on various static techniques.
2. Dynamic Analysis
   - Input A? Input B? Input C? ...
   - Based on concrete inputs to application
3. Symbolic Execution
   - Interpret the application.
   - Track "constraints" on variables.
   - When the required condition is triggered, "concretize" to obtain a possible input

Applications: Binary Rewriting



Applications: Binary Rewriting



Applications: Binary Rewriting

1. Software fault isolation (SFI)
2. Control Flow Integrity (CFI)
3. Binary code hardening
4. Binary code reuse
5. Platform-specific optimizations
6. Vulnerability fuzzing by rewriting binary code

Outline

Binary Analysis

- Binary code is everywhere, and it is the final representation of programs
- Binary analysis is challenging
- It is extremely useful to perform binary code analysis (vulnerability excavation, backdoor identification) in security
- Basic binary analysis approaches: static/dynamic analysis, symbolic execution
- There are many public available binary analysis tools