Tutorial Setup

- Interactive Session
  - Temporary shell account provided
  - Environment setup to use DyninstAPI
  - Feel free to experiment
- SSH Terminal Client
  - Login Information provided on handout
  - No SSH Terminal?
    - Google Putty
- Not a Demo
  - Got a question? Ask it.

Shell Environment

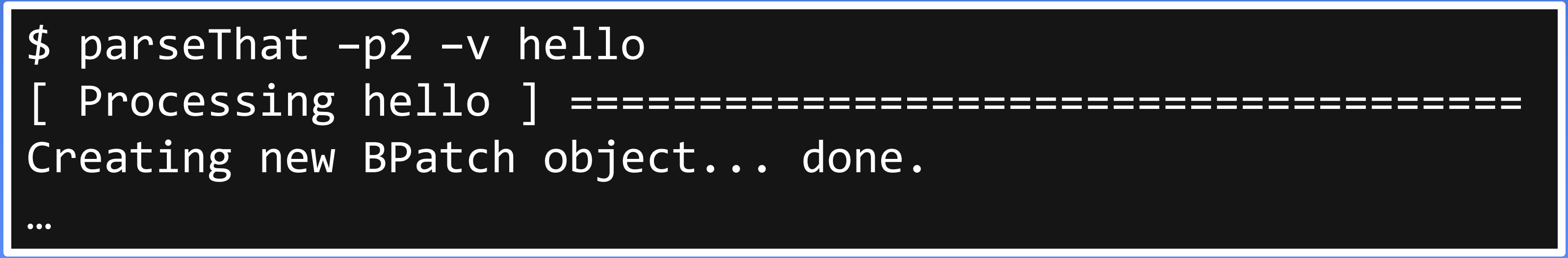
- Home Directory
  - Hello World (hello.c)
  - Quicksort (qsort.c)
  - Sample mutator (watcher.cxx)
  - Sample mutatee (caller.c)
- Shell Environment
  - LD_LIBRARY_PATH includes Dyninst libraries
  - PATH includes parseThat

Pre-Built Mutator

- parseThat
  - General tool for parsing and instrumentation
  - User-controlled depth of parsing
    - Module
    - Function
    - Control-Flow Graph
  - User-controlled depth of instrumentation
    - Function entry/exit
    - Basic blocks
    - Memory reads/writes

Basic ParseThat (Parsing)

- Parsing depth control flag (-p)
  - Module (-p0)
  - Function (-p1)
  - Control-Flow Graph (-p2)
- Depth flag is not absolute
  - Deeper parsing will occur on-demand if needed



```
$ parseThat -p2 -v hello
[ Processing hello ] =====================================
Creating new BPatch object... done.
…
```

  - Add the –v flag to see additional information

Basic ParseThat (Instrumentation)

- Default instrumentation
  - Mutatee allocates heap memory for counter
  - Increment new memory at specific locations
- Instrumentation control flag (-i)
  - Function entry (-i1)
  - Function exit (-i2)
  - Basic block (-i3)
  - Memory read instruction (-i4)
  - Memory write instruction (-i5)
- Event report flag (-s)
  - Instrument the mutatee to print

Intermediate ParseThat

- Call tracing (-T)
  - Print a message at function entry points
  - Use integer argument to limit output
    - -T=10 only prints last 10 function calls

```
$ parseThat –i1 -T qsort 20
[ Processing qsort ] =====================================
Creating new BPatch object... done.
…
```

- Useful for retrieving final call path of crashing programs

Advanced ParseThat

- Additional features
  - Attach to running program
  - Write instrumented binary to disk
  - Selective instrumentation
    - Use regular expressions to choose functions
  - Load your own instrumentation library
    - Shared libraries loaded
  - Track memory/cpu resource usage
    - Used for our nightly tests

Analysis of Malicious Software

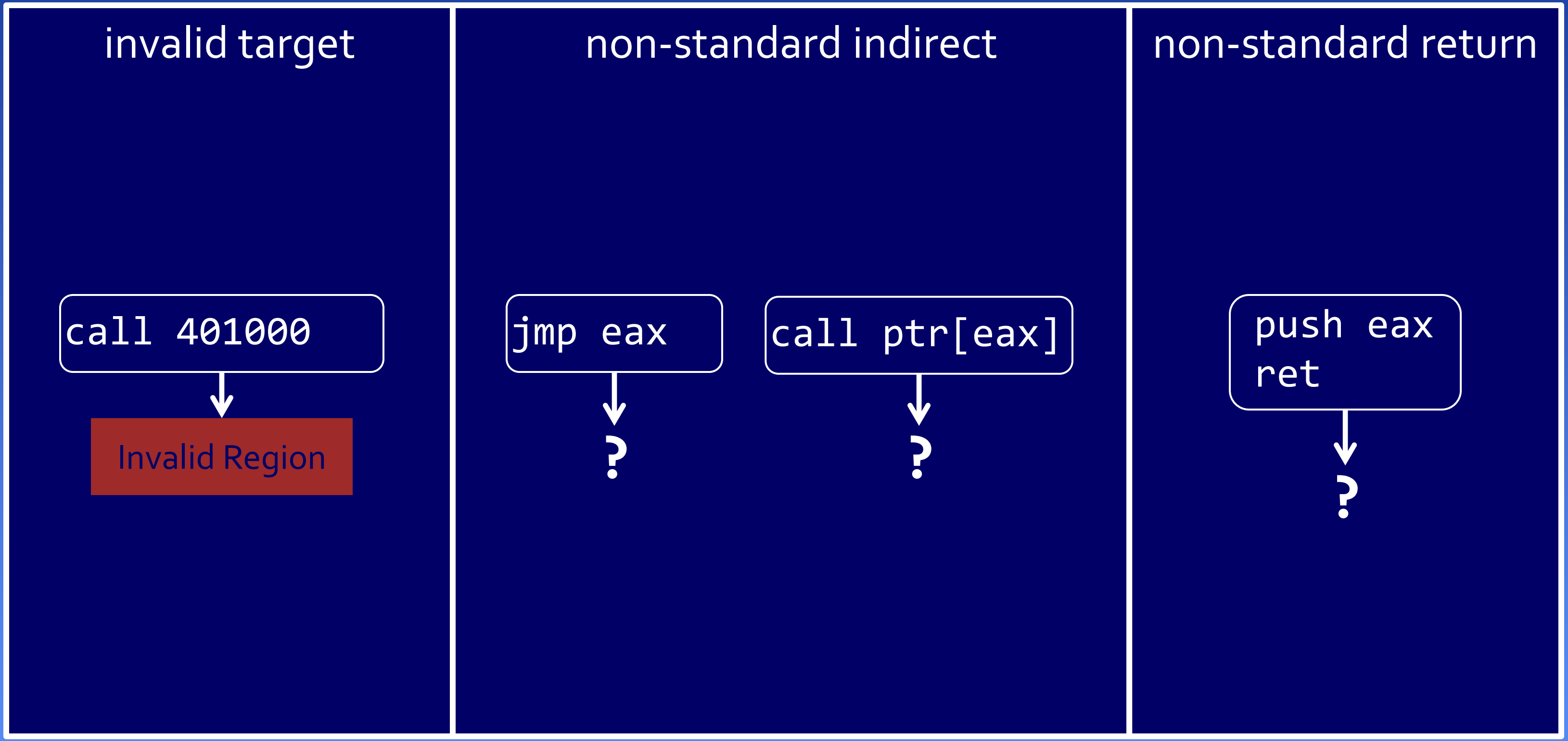
Why malware?

- Malware attacks cost billions of dollars annually[1][2]
- 28 days on average to resolve a cybercrime[2]
- 90% of malware resists analysis[3]

[1] Computer Economics. 2007 [2] Norton. 2010 [3] McAfee. 2008

Slides adapted from
Kevin Roundy <roundy@cs.wisc.edu>

Unresolvable Control-Flow



invalid target

```
call 401000
```

Invalid Region

non-standard indirect

```
jmp eax
```

?

```
call ptr[eax]
```

?

non-standard return

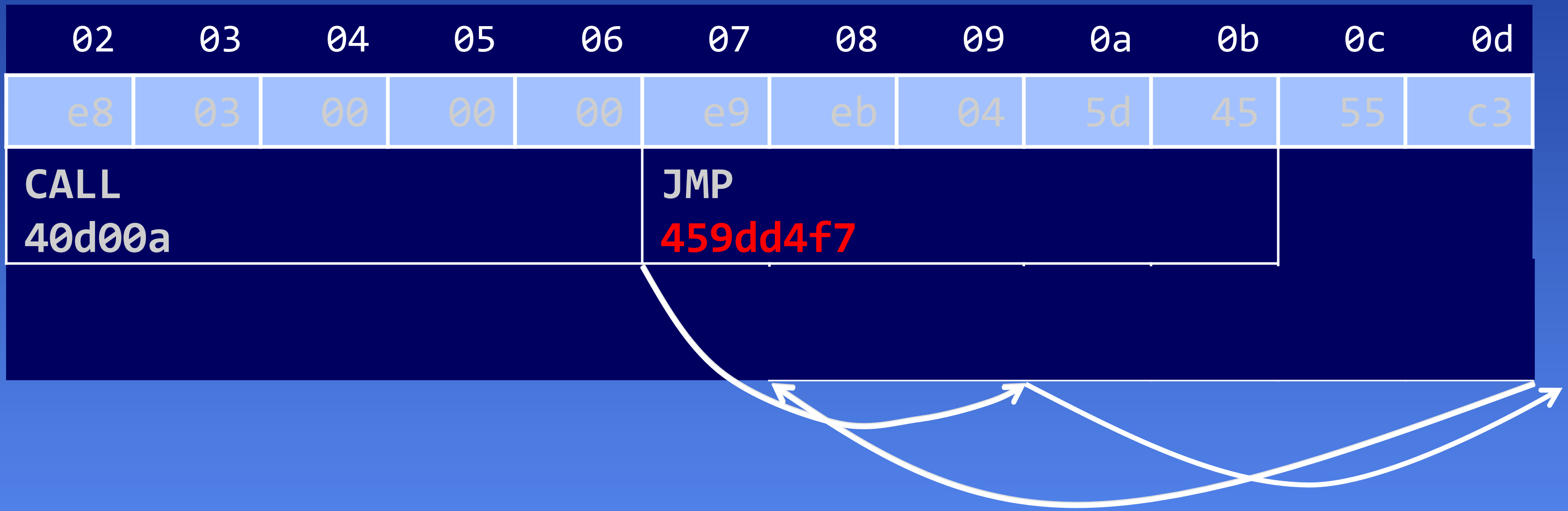
```
push eax
ret
```

?

Slides adapted from
Kevin Roundy <roundy@cs.wisc.edu>

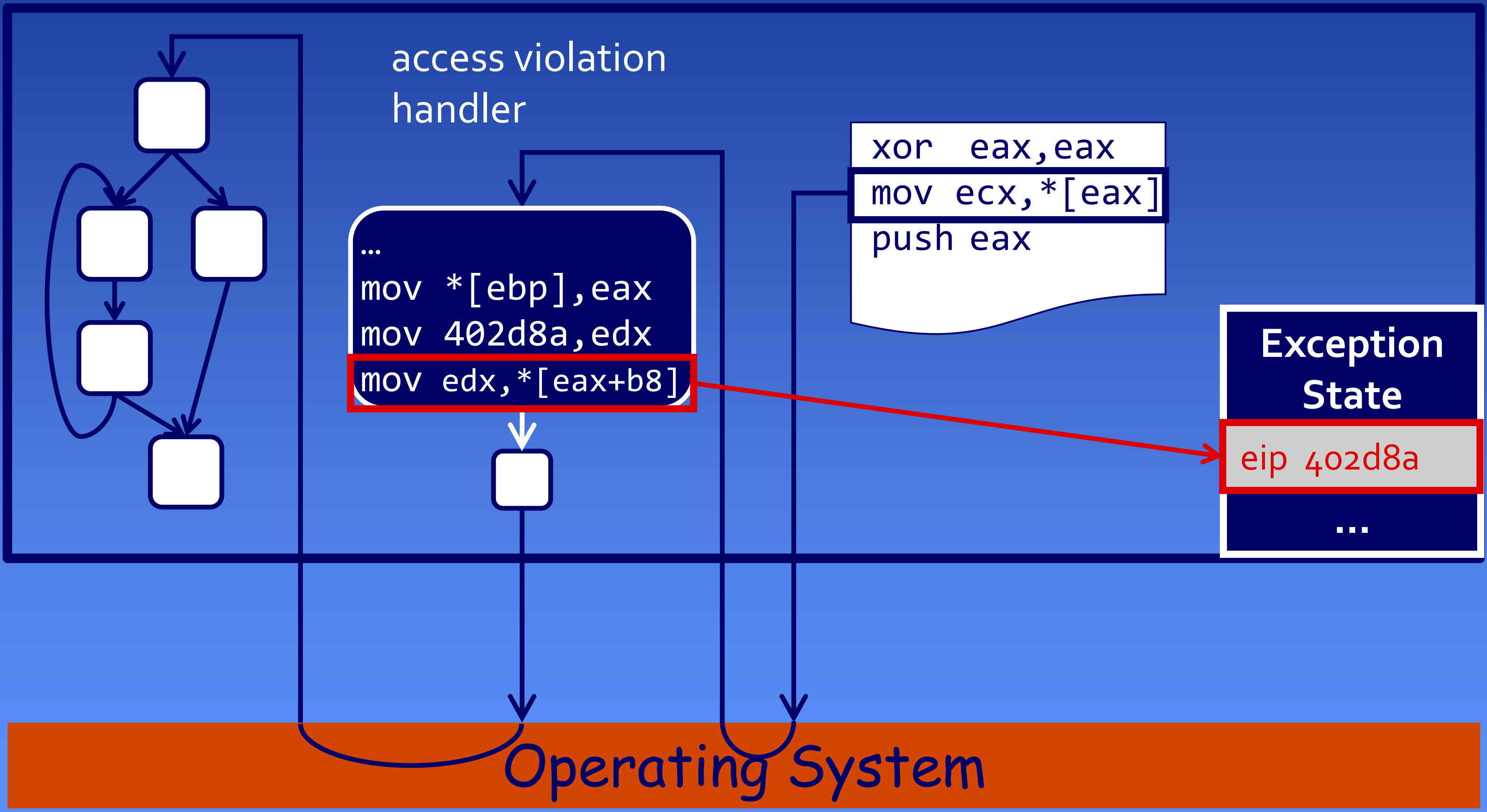
Call-Stack Tampering

Base address: 0x40d002



02	03	04	05	06	07	08	09	0a	0b	0c	0d
e8	03	00	00	00	e9	eb	04	5d	45	55	c3
CALL 40d00a					JMP 459dd4f7						

Exception-based Control-Flow

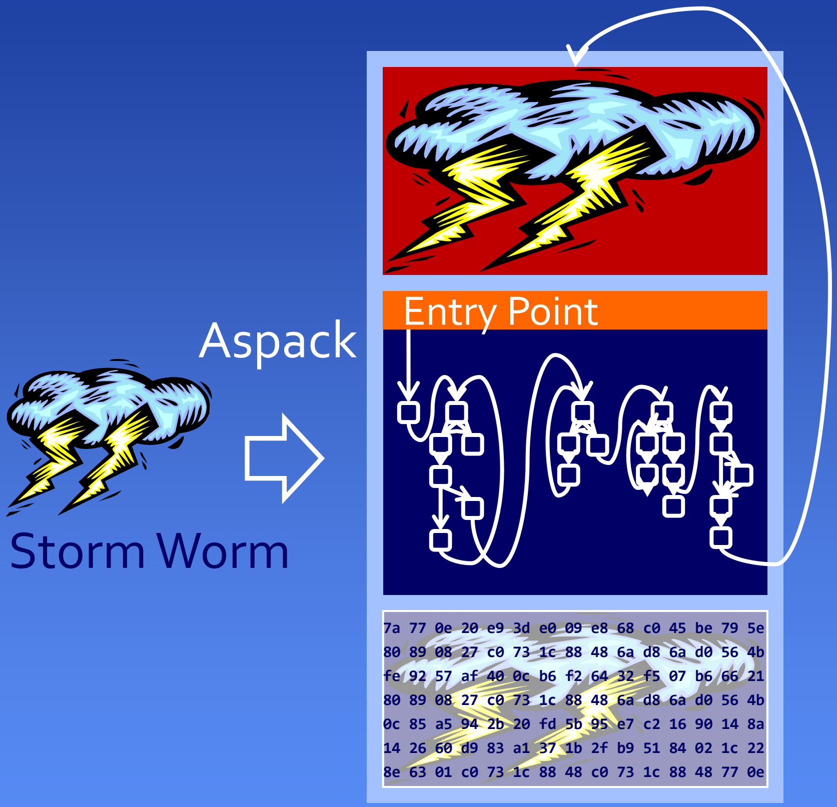


access violation handler

```
…
mov  *[ebp],eax
mov  402d8a,edx
mov  edx,*[eax+b8]
```

```
xor  eax,eax
mov  ecx,*[eax]
push eax
```

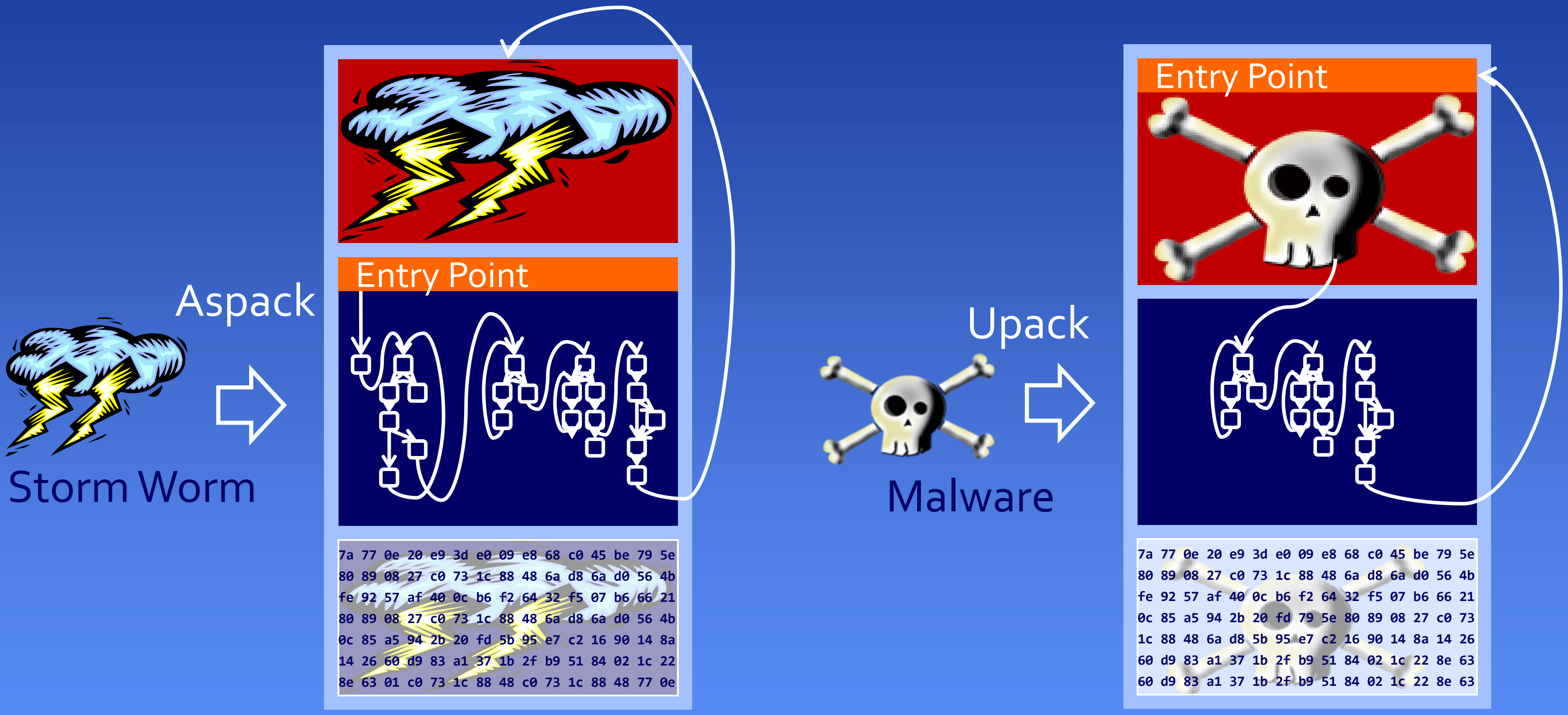
Code Packing



```
7a 77 0e 20 e9 3d e0 09 e8 68 c0 45 be 79 5e
80 89 08 27 c0 73 1c 88 48 6a d8 6a d0 56 4b
fe 92 57 af 40 0c b6 f2 64 32 f5 07 b6 66 21
80 89 08 27 c0 73 1c 88 48 6a d8 6a d0 56 4b
0c 85 a5 94 2b 20 fd 5b 95 e7 c2 16 90 14 8a
14 26 60 d9 83 a1 37 1b 2f b9 51 84 02 1c 22
8e 63 01 c0 73 1c 88 48 c0 73 1c 88 48 77 0e
```

Slides adapted from
Kevin Roundy <roundy@cs.wisc.edu>

Code Overwriting

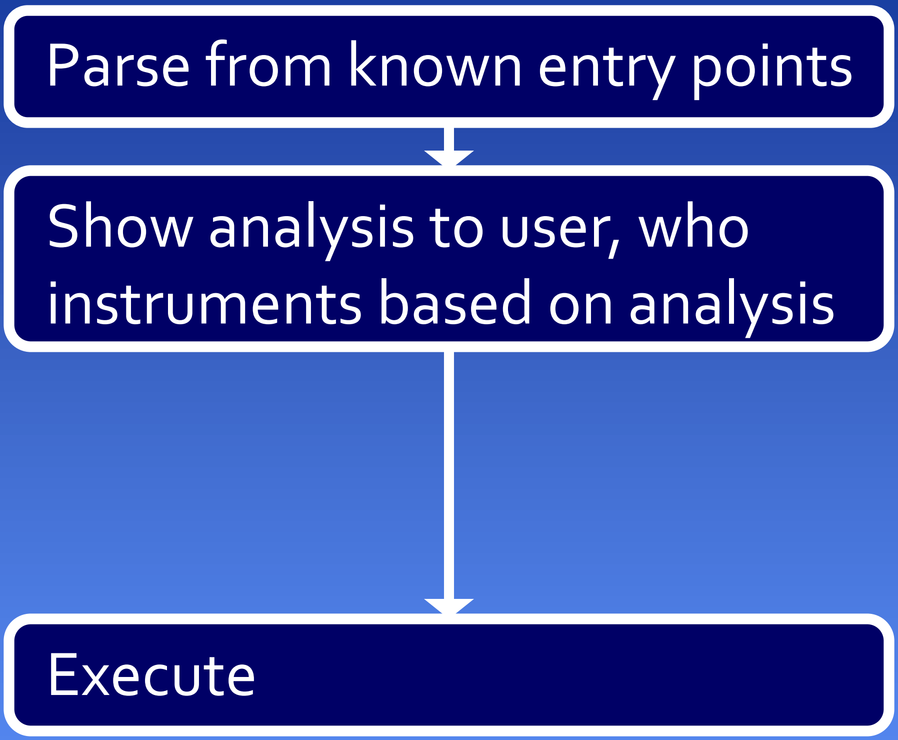


```
7a 77 0e 20 e9 3d e0 09 e8 68 c0 45 be 79 5e
80 89 08 27 c0 73 1c 88 48 6a d8 6a d0 56 4b
fe 92 57 af 40 0c b6 f2 64 32 f5 07 b6 66 21
80 89 08 27 c0 73 1c 88 48 6a d8 6a d0 56 4b
0c 85 a5 94 2b 20 fd 5b 95 e7 c2 16 90 14 8a
14 26 60 d9 83 a1 37 1b 2f b9 51 84 02 1c 22
8e 63 01 c0 73 1c 88 48 c0 73 1c 88 48 77 0e
```

```
7a 77 0e 20 e9 3d e0 09 e8 68 c0 45 be 79 5e
80 89 08 27 c0 73 1c 88 48 6a d8 6a d0 56 4b
fe 92 57 af 40 0c b6 f2 64 32 f5 07 b6 66 21
0c 85 a5 94 2b 20 fd 79 5e 80 89 08 27 c0 73
1c 88 48 6a d8 5b 95 e7 c2 16 90 14 8a 14 26
60 d9 83 a1 37 1b 2f b9 51 84 02 1c 22 8e 63
60 d9 83 a1 37 1b 2f b9 51 84 02 1c 22 8e 63
```

Slides adapted from
Kevin Roundy <roundy@cs.wisc.edu>

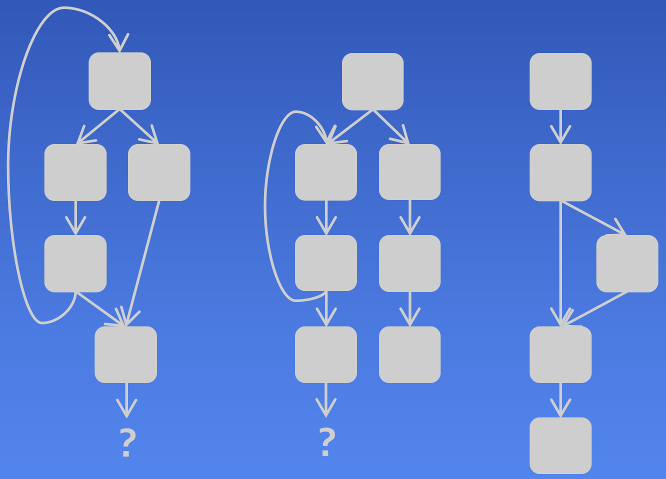
Static Analysis Only



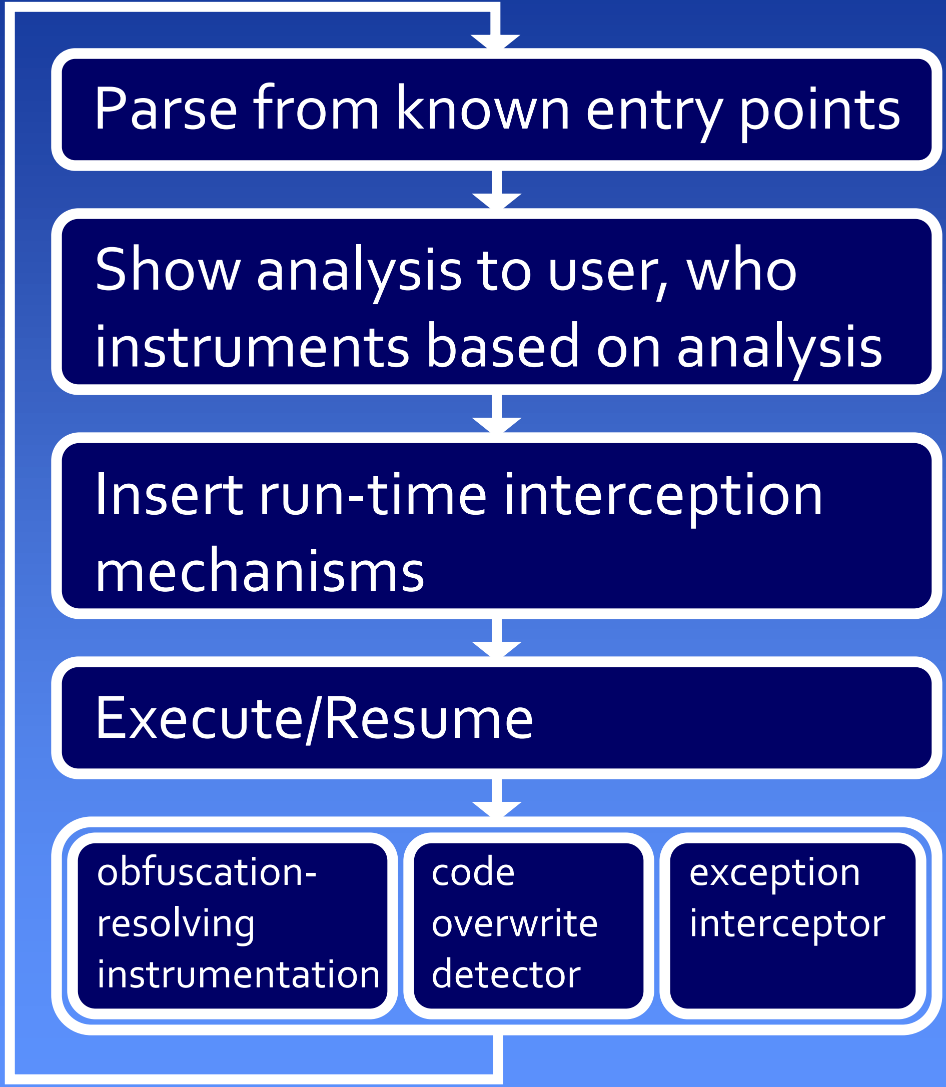
Parse from known entry points

Show analysis to user, who instruments based on analysis

Execute



Static/Dynamic Hybrid Analysis



Parse from known entry points

Show analysis to user, who instruments based on analysis

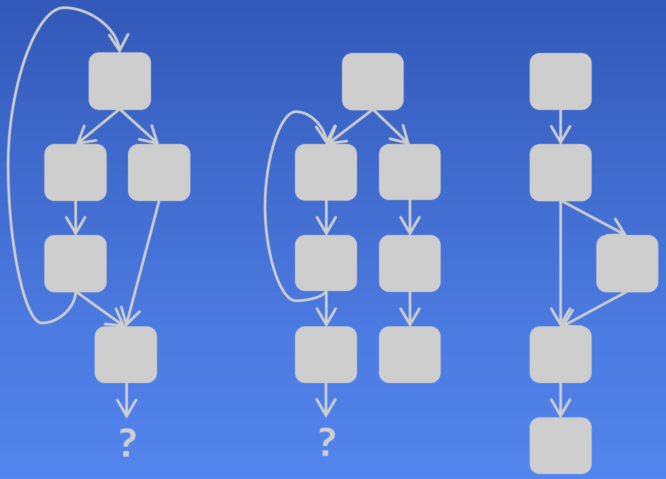
Insert run-time interception mechanisms

Execute/Resume

obfuscation-resolving instrumentation

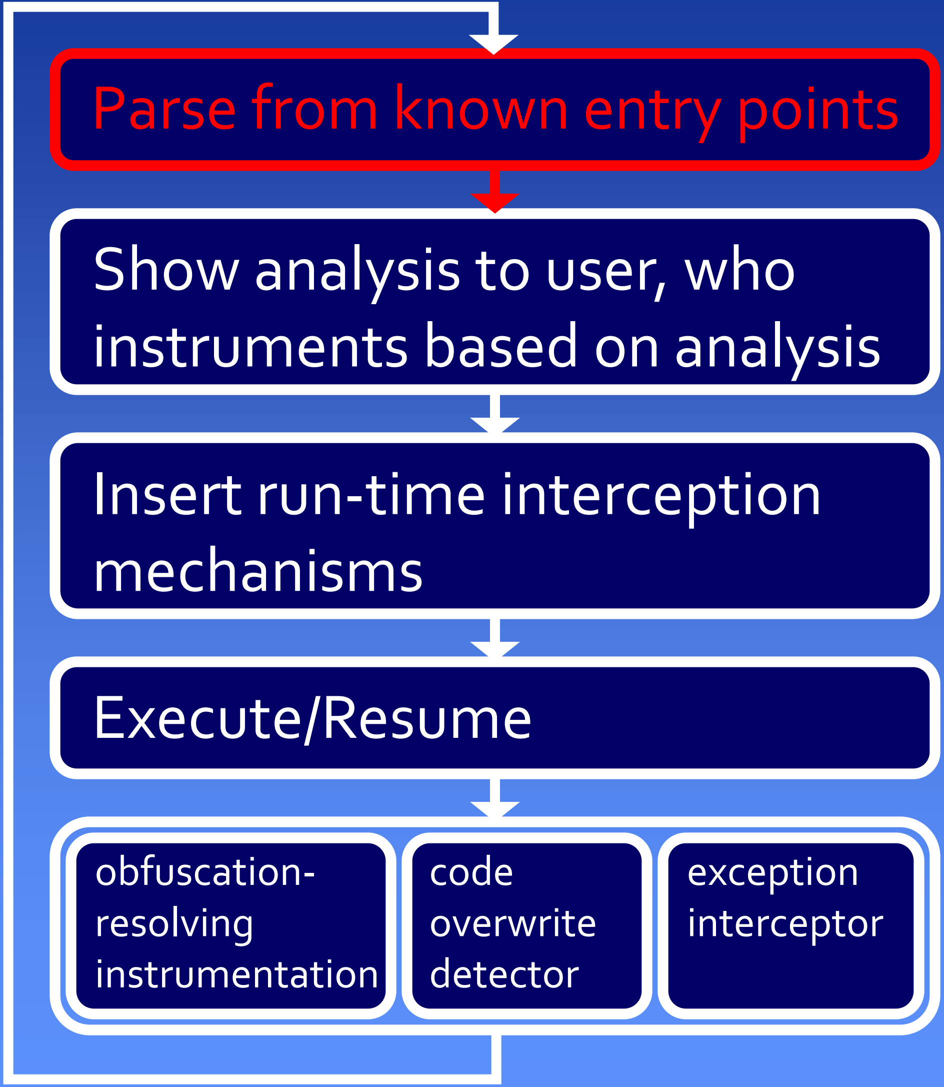
code overwrite detector

exception interceptor



Slides adapted from
Kevin Roundy <roundy@cs.wisc.edu>

Static/Dynamic Hybrid Analysis



Parse from known entry points

Show analysis to user, who instruments based on analysis

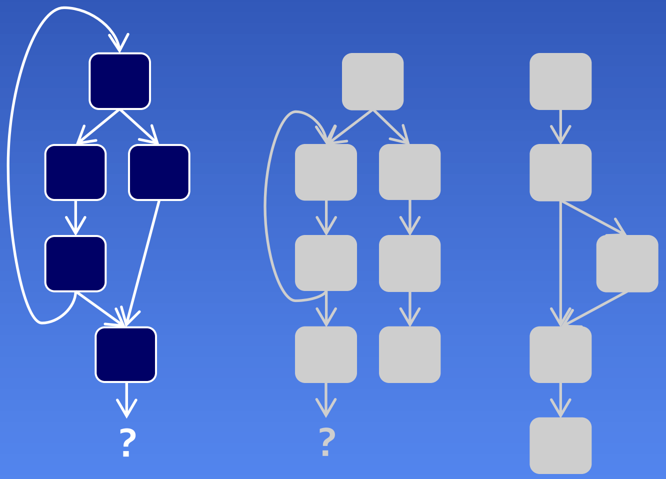
Insert run-time interception mechanisms

Execute/Resume

obfuscation-resolving instrumentation

code overwrite detector

exception interceptor

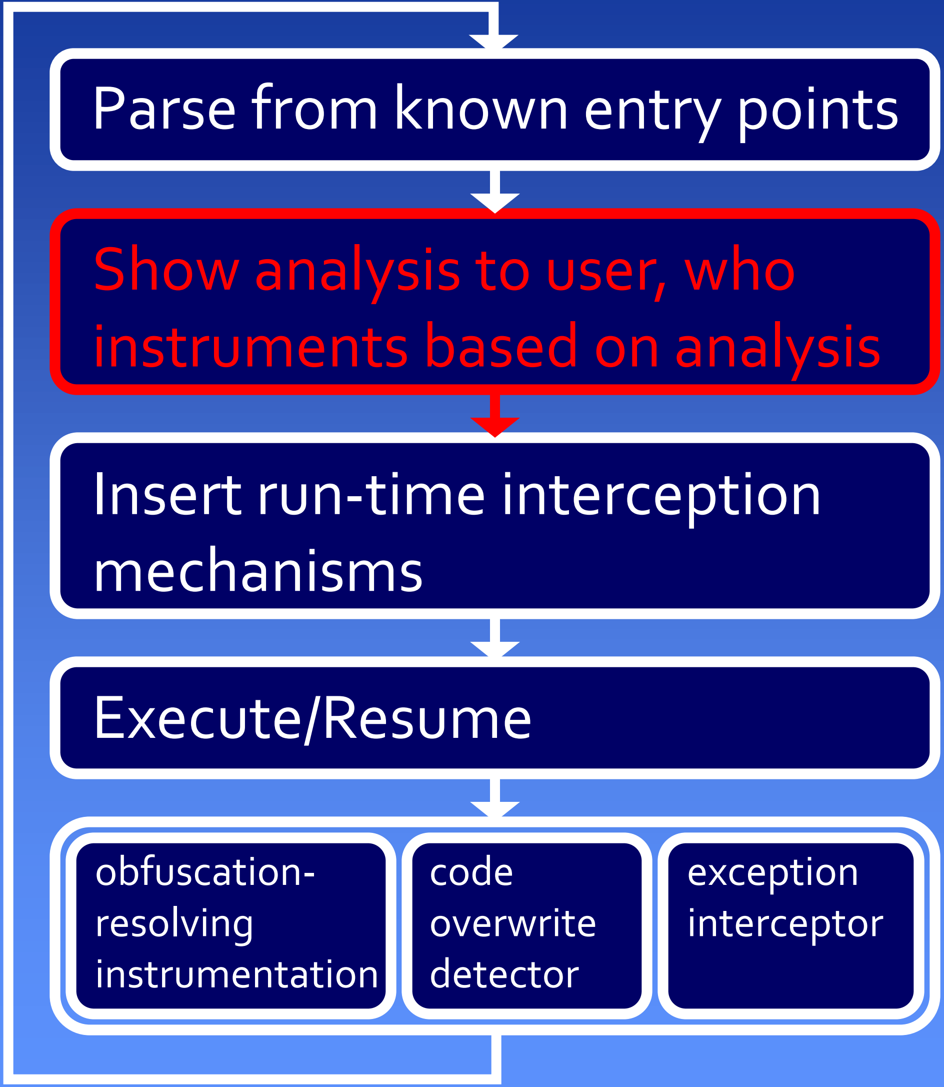


Slides adapted from
Kevin Roundy <roundy@cs.wisc.edu>

University of Maryland

Dyn inst

Static/Dynamic Hybrid Analysis



Parse from known entry points

Show analysis to user, who instruments based on analysis

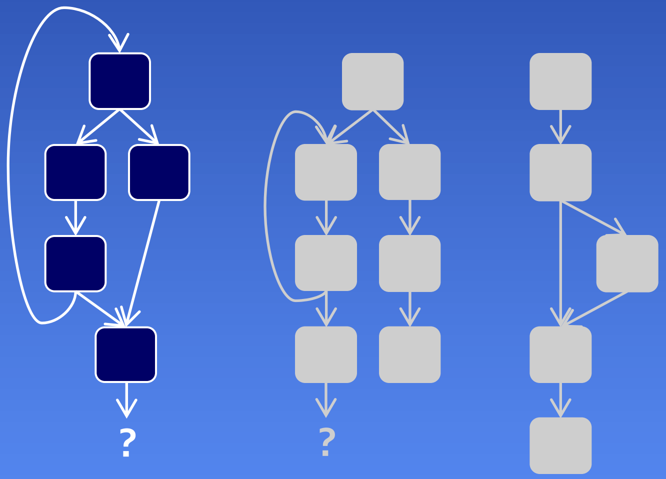
Insert run-time interception mechanisms

Execute/Resume

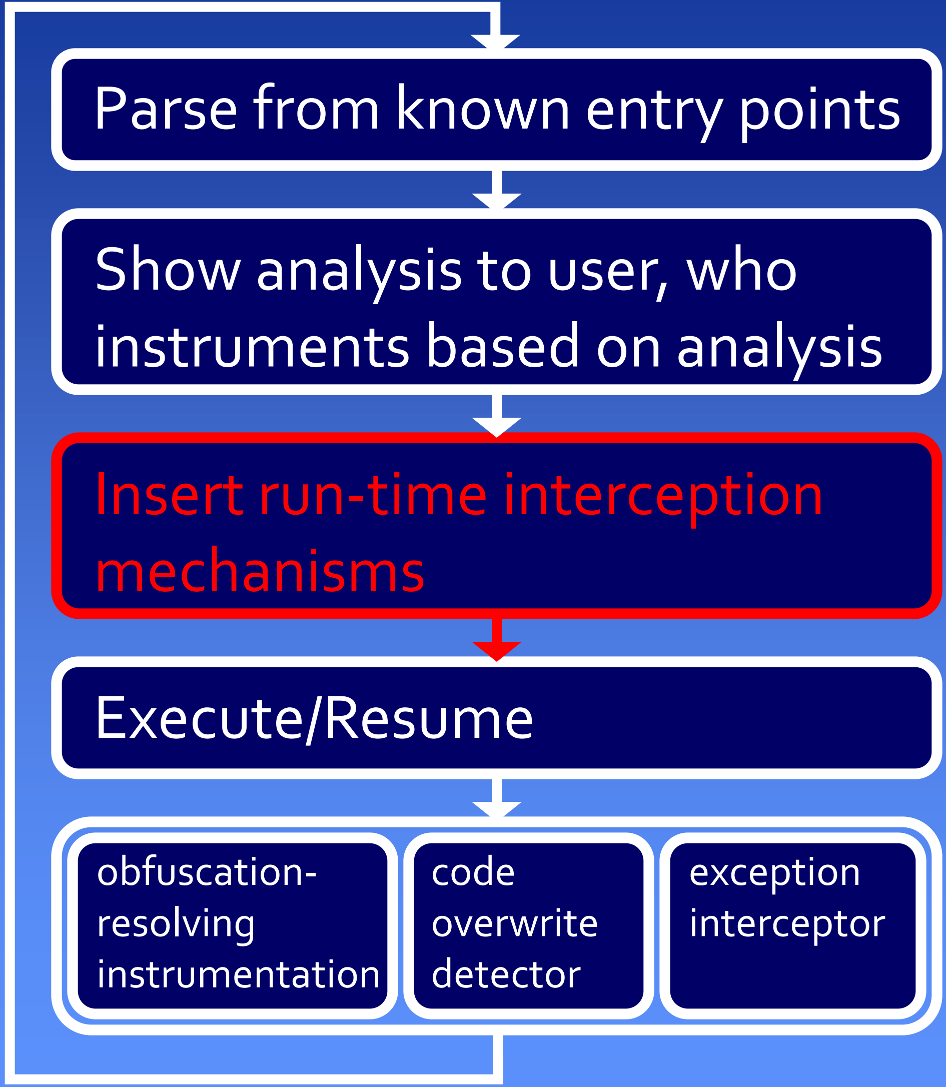
obfuscation-resolving instrumentation

code overwrite detector

exception interceptor



Static/Dynamic Hybrid Analysis



Parse from known entry points

Show analysis to user, who instruments based on analysis

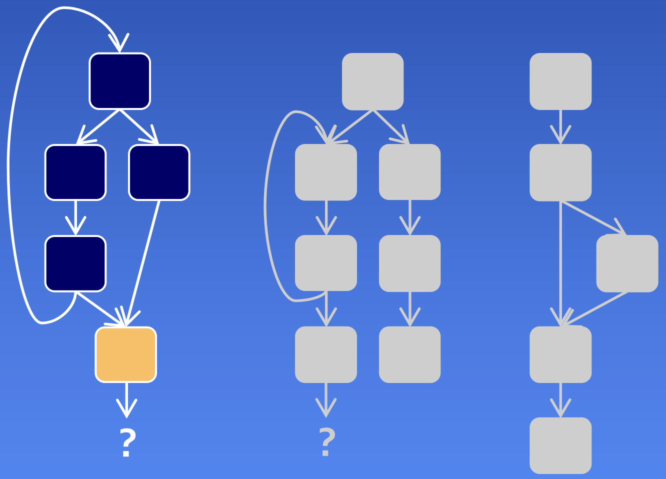
Insert run-time interception mechanisms

Execute/Resume

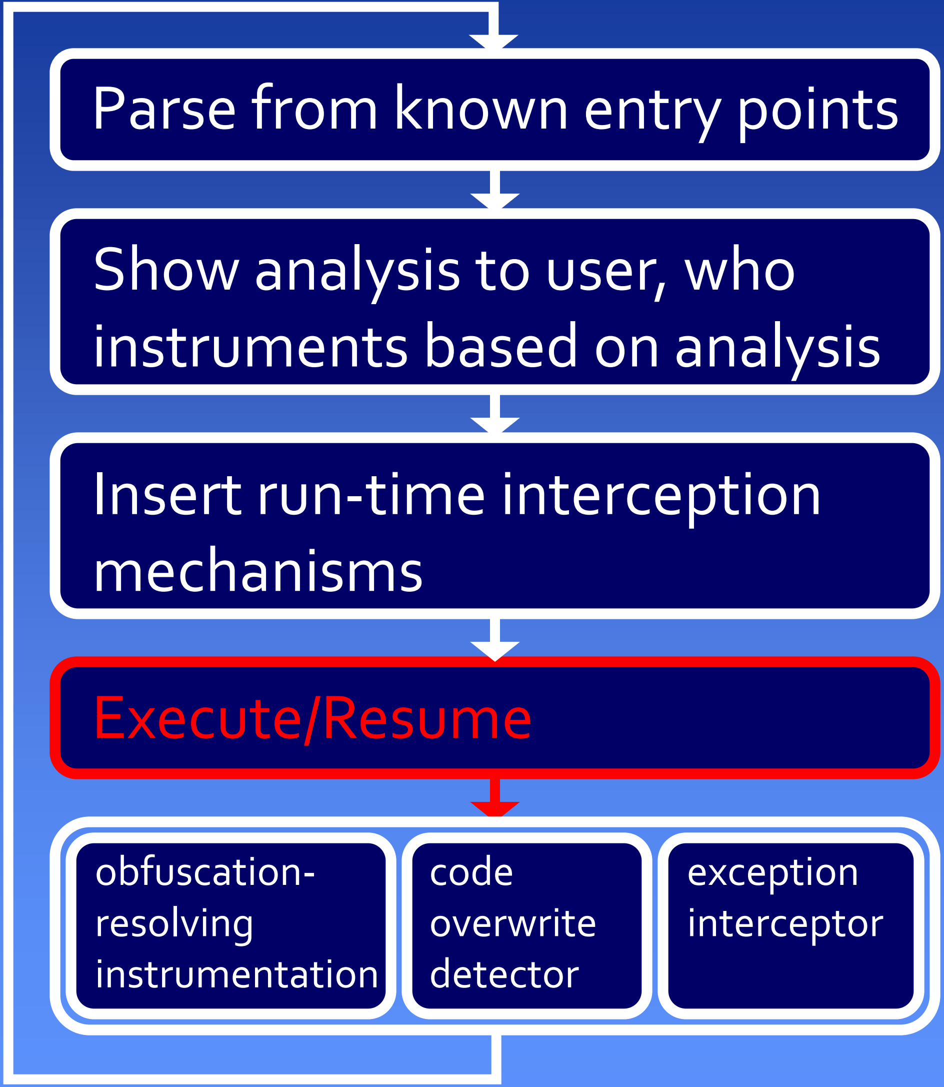
obfuscation-resolving instrumentation

code overwrite detector

exception interceptor



Static/Dynamic Hybrid Analysis



Parse from known entry points

Show analysis to user, who instruments based on analysis

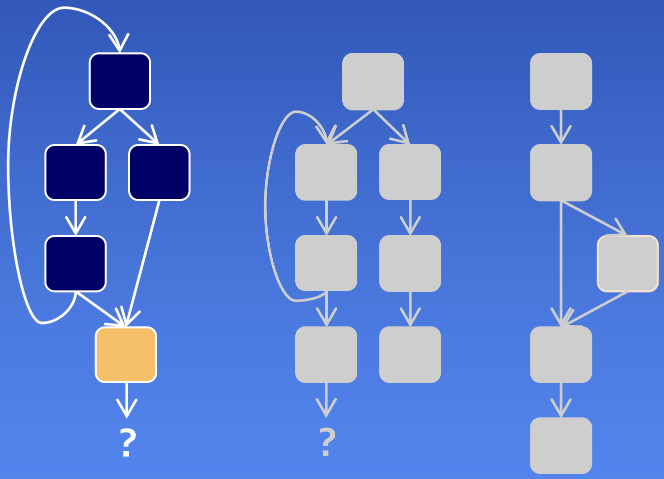
Insert run-time interception mechanisms

Execute/Resume

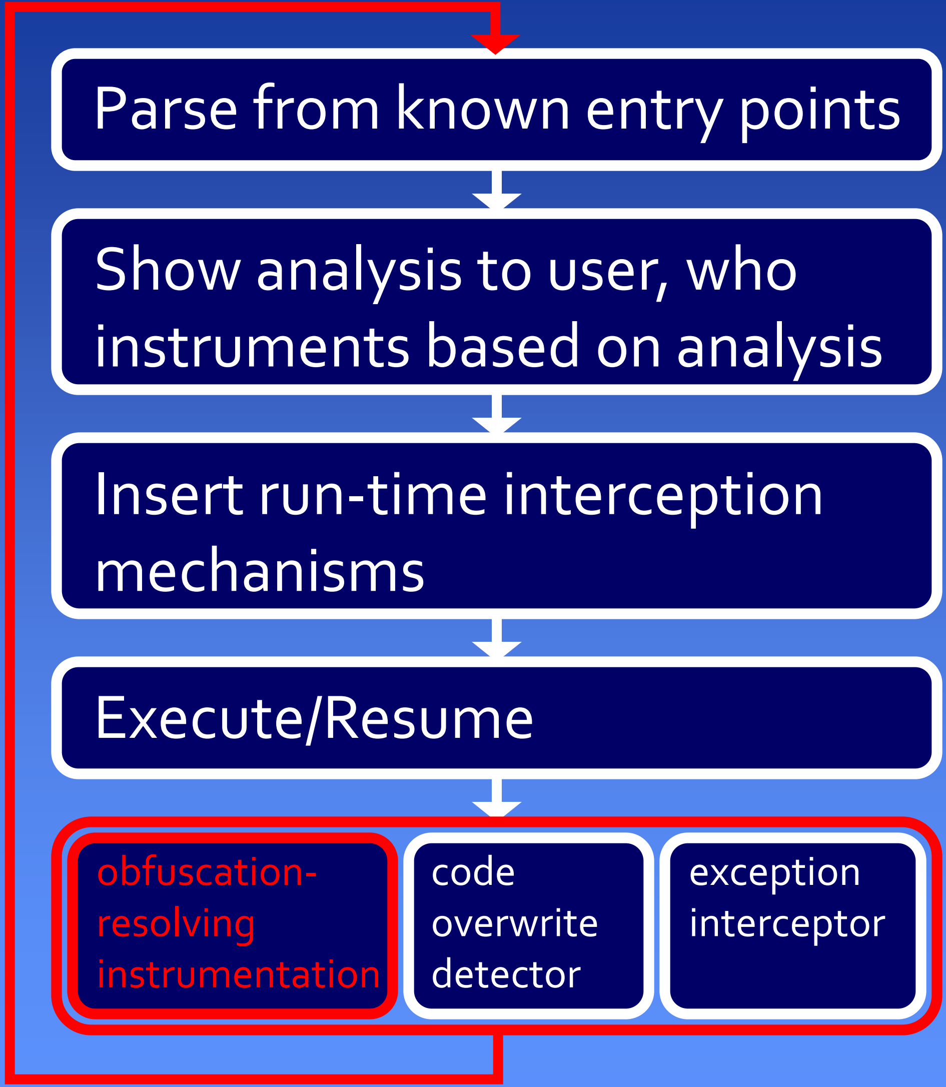
obfuscation-resolving instrumentation

code overwrite detector

exception interceptor



Static/Dynamic Hybrid Analysis



Parse from known entry points

Show analysis to user, who instruments based on analysis

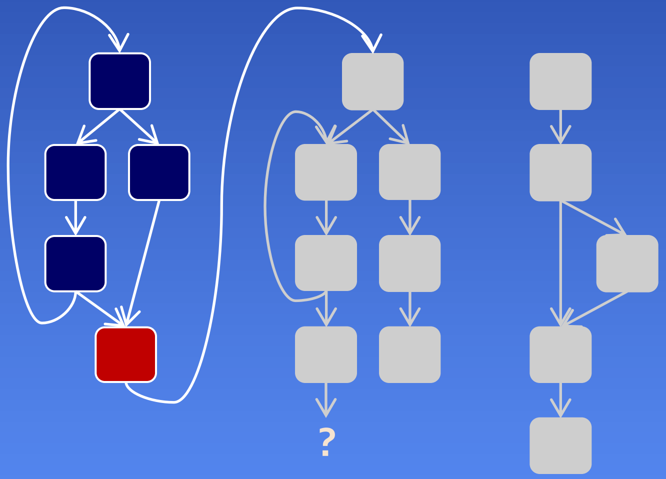
Insert run-time interception mechanisms

Execute/Resume

obfuscation-resolving instrumentation

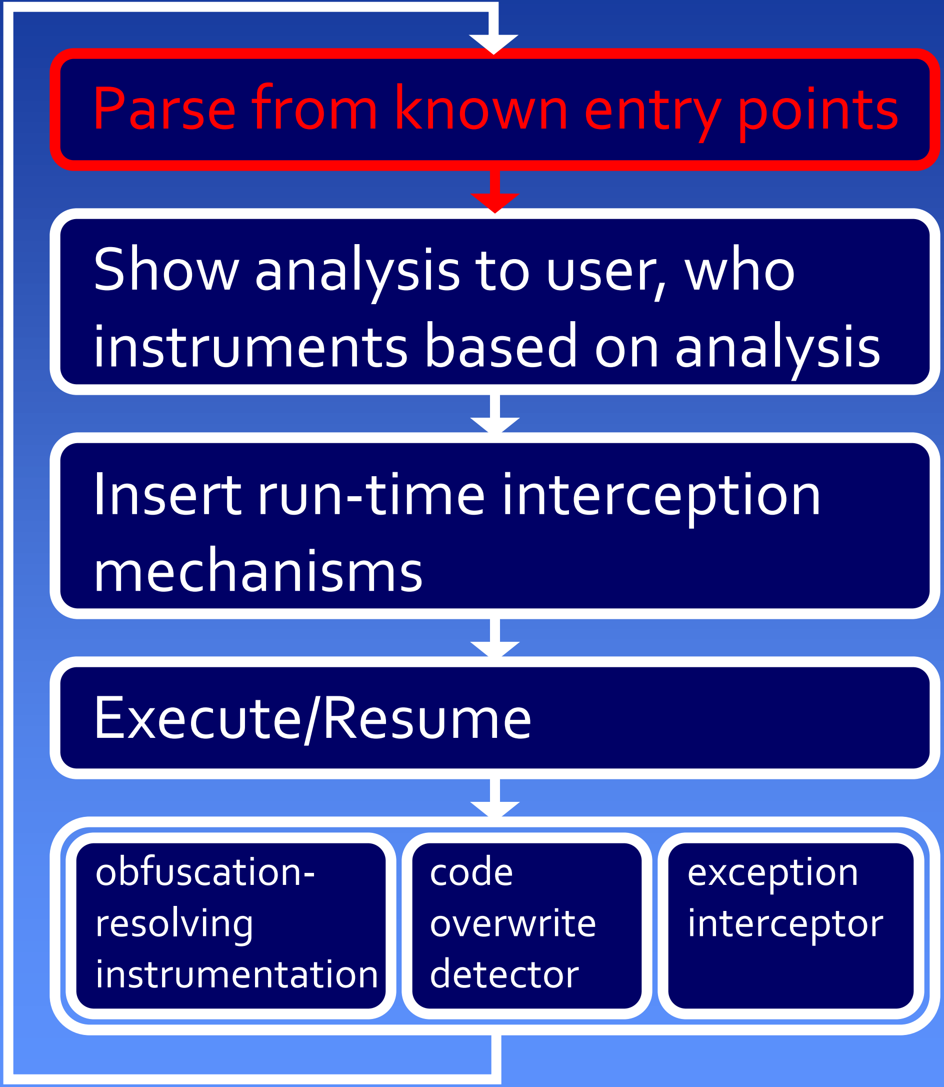
code overwrite detector

exception interceptor



Slides adapted from
Kevin Roundy <roundy@cs.wisc.edu>

Static/Dynamic Hybrid Analysis



Parse from known entry points

Show analysis to user, who instruments based on analysis

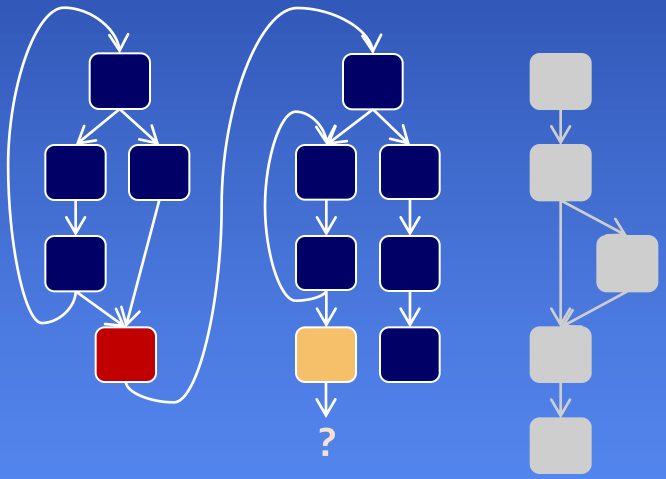
Insert run-time interception mechanisms

Execute/Resume

obfuscation-resolving instrumentation

code overwrite detector

exception interceptor



Slides adapted from
Kevin Roundy <roundy@cs.wisc.edu>

Our Simple Malware Mutator

- Dyninst provides the functionality
  - Kevin Roundy
  - Beyond the scope of this tutorial
- Unresolvable control-flow watcher
  - Statically analyze binary for the following:
    - Function entry points
    - Dynamic call points
  - Maintain a set of function entry addresses
  - Pause mutatee at dynamic call points mid-run
    - Check target address against function entry
    - If invalid, kill the mutatee