Dyninst: A Binary Analysis and Modification Framework

Jeffrey K. Hollingsworth

Ray Chen

University of Maryland

Department of Computer Science

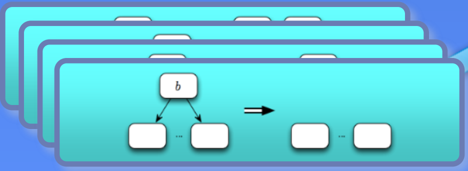
Binary modification

Binary Program

```
1d8d481674c08548530033
0019058b48854808c38348
08438b48d0ff0033000c00
00441f0f660000441f0fc3
5bf175c01d8d481674c085
48530032ff1058b4885480
8c3834808438b48d0ff003
2ffc490909090909090909
09090c35bf175c00000000
0801f0f0000000801f0fc
3f30000000801f0f00014
427e808ec8348
```

Binary Modification Toolkit

Modification Requests



Behavior Analysis

Attack Detection

Optimization

Performance Analysis

Modified Binary Program

```
1d8d481674c08548530033
0019058b48854808c38348
b1eb000001003337a205c
00441f0f660000441f0fc3
5bf175c01d8d481674c085
f82474894cf0246c894ce
8c3834808438b48d0ff003
2ffc490909090909090909
64894ccd8948d8245c894
0801f0f0000000801f0fc
3f30000000801f0f00014
fab70f087448503966003
```

Fault Diagnosis

Cyberforensics

Testing

Debugging

Simulation

Program Auditing

Dynamic (“Hot”) Patching

Uses For Runtime Code Patching

- Security & Testing
  - Code coverage testing
  - Monitoring (dynamic taint analysis)
- Correctness debugging
  - Fast conditional breakpoints
  - Data breakpoints
- Execution driven simulation
  - Architecture studies

Why Binary Analysis and Manipulation?

- It's what runs on the computer
- All compiled languages (more or less) look the same as a binary
- No Source Code Required
  - For commercial and malware, often not available
- Implicitly Picks up compiler issues
  - Security problems due to compiler bugs

What is Dyninst?

- API for
  - binary analysis
  - binary re-writing
  - runtime patching
- Features
  - Generates info about the binary
    - Example: Recover control flow graphs
  - New code can be added to programs during execution
    - Permits instrumentation and modification
  - Provides processor independent abstractions
  - Platform independent patching
    - API abstracts away OS, hardware differences

Dyninst Design Philosophy

- Use Any Data Available
  - Debug symbols
  - Dynamic Linker info
  - Binary Analysis within Dyninst
  - User Supplied Info
- Work when any source of data is missing
  - Stripped binaries
  - Static linked program
  - Obfuscated binaries

Type & Variable Support in Dyninst

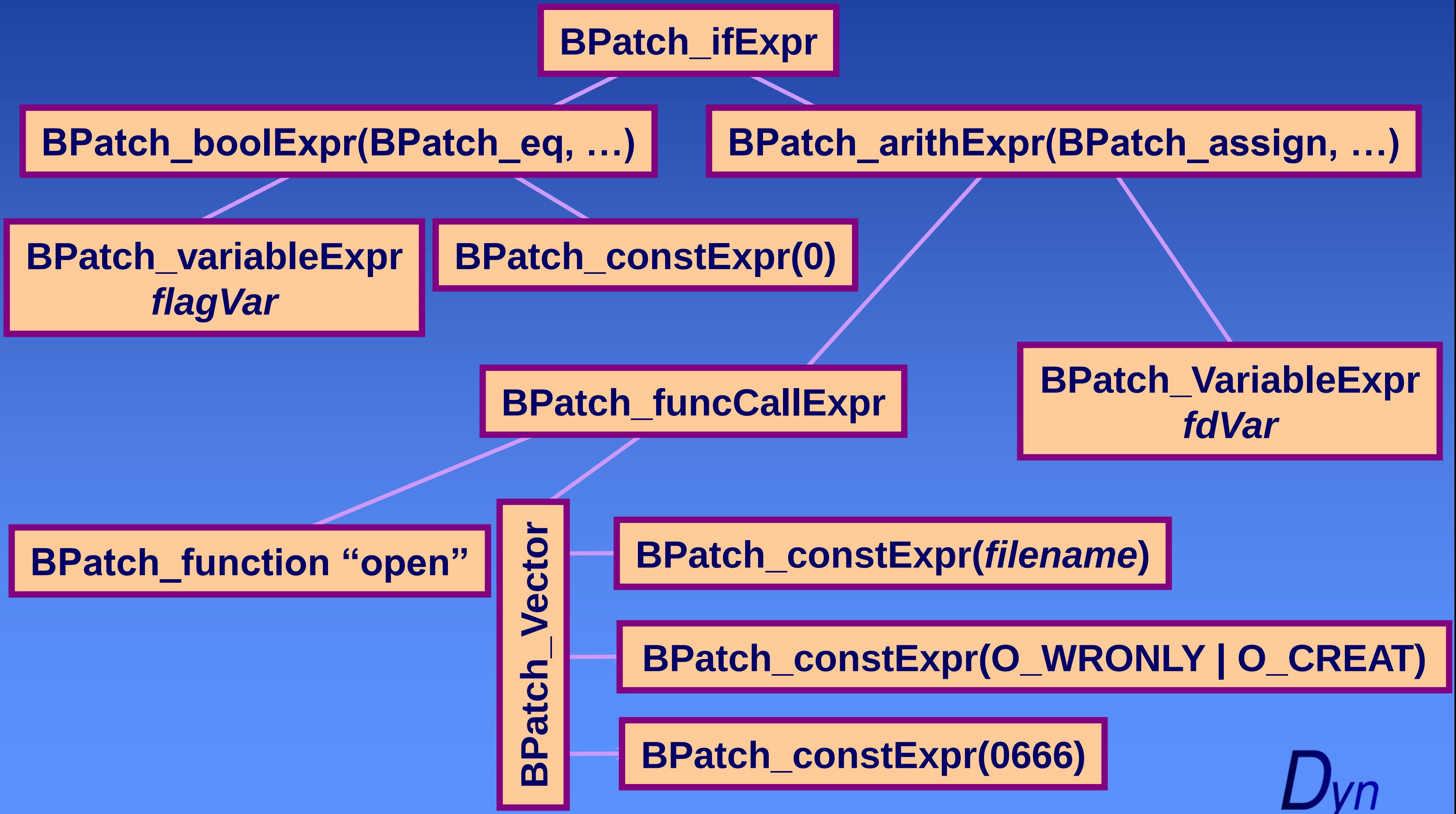
- Access to local (stack) variables
- Complex types
  - non-integer scalars
  - structures
  - arrays
  - Fortran common blocks
- Example: Correctness debugging
  - print contents of data structures

Representing New Code Snippets

- Platform Independent Representation
  - Same code can be inserted into apps on any system
- Simple Abstract Syntax Tree
  - Can refer to application state (variables & params)
  - Includes simple looping construct
  - Permits calls to application subroutines
- Type Checking
  - Ensures that snippets are type compatible
  - Based on structural equivalence
    - allows flexibility when adding new code

Snippet Example

if (flagVar == 0) fdVar = open(filename, ...)



BPatch_ifExpr

BPatch_boolExpr(BPatch_eq, …)

BPatch_arithExpr(BPatch_assign, …)

BPatch_variableExpr
flagVar

BPatch_constExpr(0)

BPatch_funcCallExpr

BPatch_VariableExpr
fdVar

BPatch_function “open”

BPatch_Vector

BPatch_constExpr(*filename*)

BPatch_constExpr(O_WRONLY | O_CREAT)

BPatch_constExpr(0666)

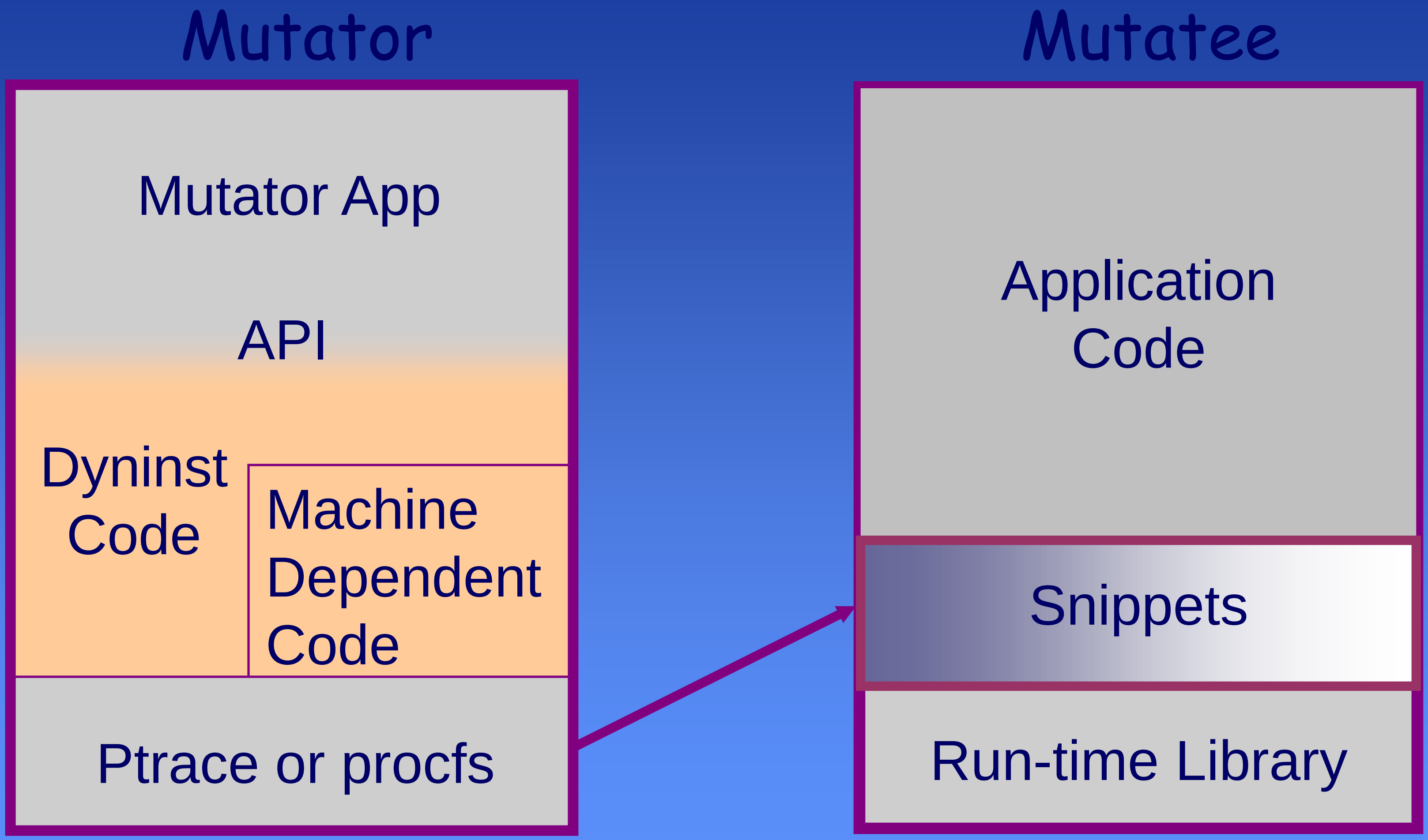
Memory Instrumentation

- Dynamic memory access instrumentation
  - collect low level memory accesses
  - with the flexibility of dynamic instrumentation
- Possible applications
  - tools to catch memory errors
  - offline performance analysis (Sigma etc.)
  - online optimization

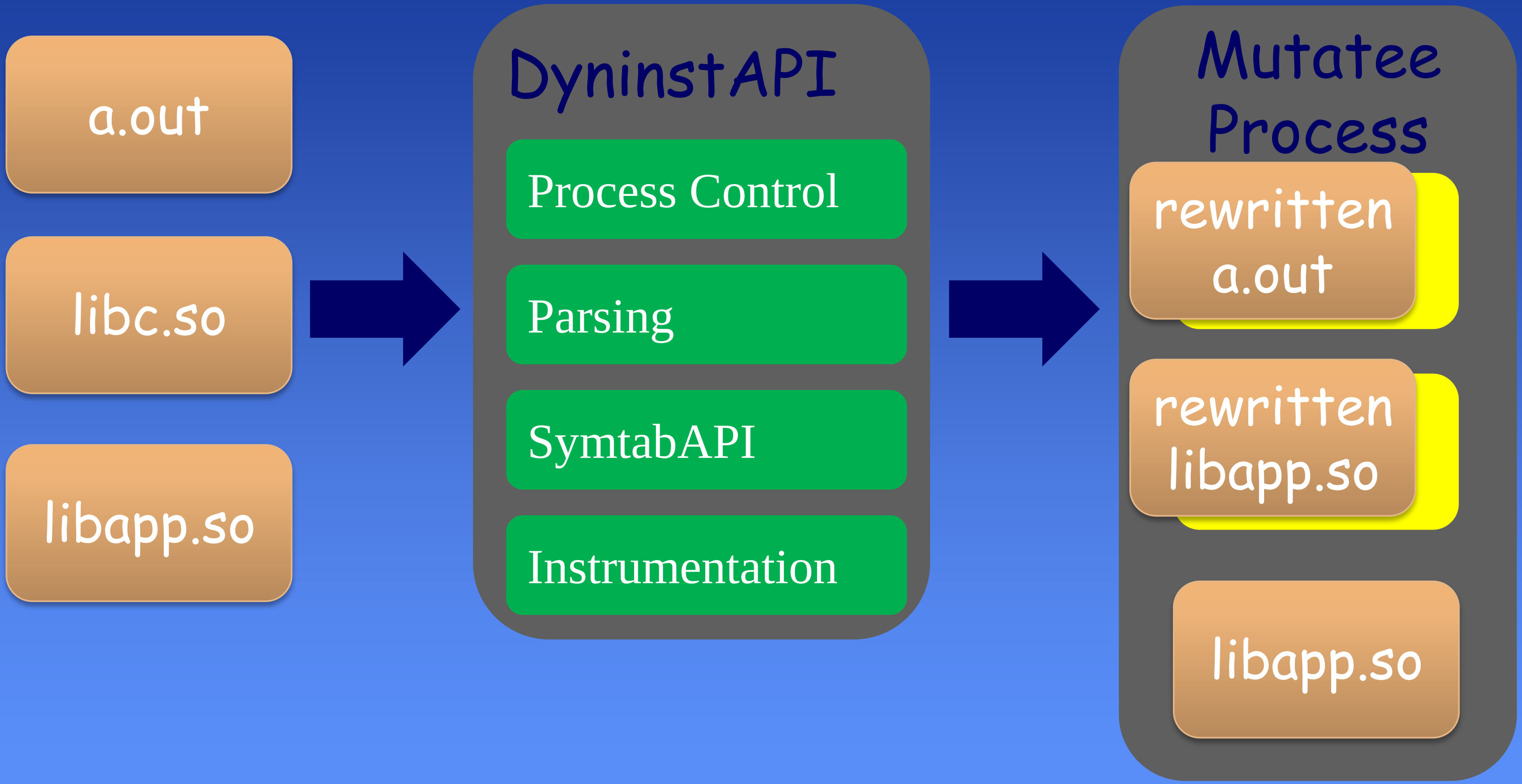
Memory Instrumentation Features

- Finding memory access instructions
  - loads, stores, prefetches
- Builds on Arbitrary Instrumentation
- Decoded instruction information
  - type of instruction
  - constants and registers involved in computing
    - the effective address
    - the number of bytes moved
  - available in the mutator before execution
- Memory access snippets
  - effective address in process space
  - byte count
  - available in mutatee at execution time

Runtime Binary Modification



Static Binary Rewriting in Dyninst



a.out

libc.so

libapp.so

DyninstAPI

Process Control

Parsing

SymtabAPI

Instrumentation

Mutatee Process

rewritten a.out

rewritten libapp.so

libapp.so

A Static Binary Rewriter

- Binary Rewriter Capabilities
  - Instrument once, run many times
  - Run instrumented binaries on systems without dynamic instrumentation (e.g. some embedded systems).
  - Perform static analysis without running a binary

- Operates on unmodified binaries.
  - No debug information required
  - No linker relocations required
  - No symbols required

- Same abstractions and interfaces as online rewriter.

Static Vs. Dynamic Rewriting

Static Rewriting

✓Faster instrumentation insertion.

✓Amortize parsing and instrumentation time across multiple runs.

✓Easier to port.

Dynamic Instrumentation

✓Insert and Remove instrumentation at run time.

✓Execute instrumentation at a particular time (oneTimeCode).

✓Respond to run time events (shared library loads, exec, …).

BPatch_addressSpace

- Use `BPatch_addressSpace` for static and dynamic code instrumentation.

```
if (use_bin_edit)
   addr_space = bpatch.openFile(...);
else
   addr_space = bpatch.attachProcess(...);

...

addr_space->getImage()->findFunction(...);
addr_space->insertSnippet(...);
addr_space->replaceFunction(...);
```

Example Use: Rewriting Symbols Tables

- Add a function symbol to a binary:

```
/* Open a file */
Symtab *symt;
Symtab::openFile(symt, “a.out”);

/* Add Symbol */
symt->createFunction(“func1” /*name*/,
                      0x1000 /*offset*/,
                      100 /*size*/);

/* Write new binary */
symt->emit(“rewritten.out”);
```

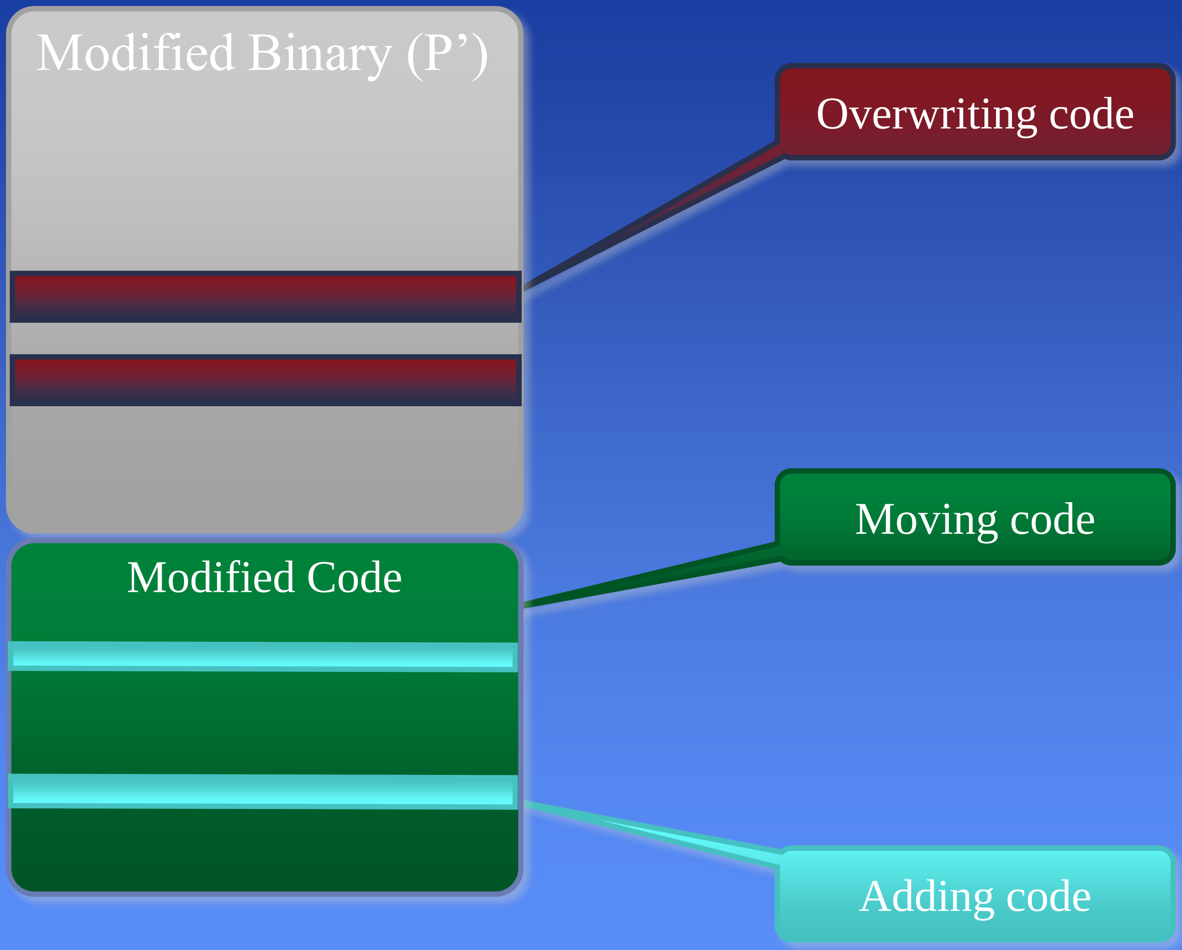
Sensitivity-resistant code relocation

- Preserve *visible behavior*
  - Relationship of input to output
- Identify *sensitive* instructions
  - Those whose behavior is changed
- Compensate for *externally sensitive* instructions
  - Those whose sensitivity affects visible behavior
- Approach
  - Binary analysis (slicing, symbolic execution)
  - Code generation
  - Runtime checks

Sensitivity

Code Replacement Actions

Effects



Modified Binary (P')

Modified Code

Overwriting code

Moving code

Adding code

Code-as-Data (CAD) Sensitive
Instructions that read or write original code

Program Counter (PC) Sensitive
Moved instructions that use the PC

Control Flow (CF) Sensitive
Instructions whose successors were moved

Allocated-vs-Unallocated (AVU) Sensitive
Instructions that test allocated memory

Example compensation transformations

PC Sensitive

```
call printf
```

→

```
push $(orig_ret_addr)
jmp printf
```

CAD/AVU Sensitive

```
mov (%eax), %ebx
```

→

```
cmp %eax, $textEnd
jge L1
mov $offset(%eax), %ebx
jmp L2
L1: mov (%eax), %ebx
L2: ...
```

Efficient group transformation (PC/CF Sensitive)

```
call ebx_thunk
ebx_thunk:
  mov (%esp), %ebx
  ret
```

→

```
mov $(ret_addr), %ebx
```

Experiments: code relocation

- Verify preservation of behavior on sensitive binaries
  - Instrument synthetic malware samples
  - Samples should execute with unchanged behavior

- Evaluate overall performance
  - Null instrumentation of SPEC CPU 2006 benchmarks, Apache, and MySQL
  - Sensitivity-resistant code relocation should reduce overhead
  - Group transformations should benefit on Apache/MySQL

Results: behavior preservation

Packer Tool	Market share	CAD sensitive	Anti-debug	Success
PolyEnE_CAD	6.21%	yes		✓
EXECryptor	4.06%	yes	yes	
Themida	2.95%	yes	yes	
PECompact_CAD	2.59%	yes		✓
ASProtect	0.43%	yes		✓
Armadillo	0.37%	yes	yes	
Yoda's Protector	0.33%	yes	yes	✓

- S-R relocation succeeded on four additional packers
- Failures are due to anti-debug techniques not yet addressed

The Dyninst Team

- Maryland
  - Jeff Hollingsworth
  - Ray Chen
  - Tugrul Ince
  - Chester Lam
  - Mike Lam
  - Geoff Stoker
  - Philip Yang
  - Yifan Zhou

- Wisconsin
  - Bart Miller
  - Bill Williams
  - Andrew Bernat
  - Michael Brim
  - Wenbin Fang
  - Emily Jacobson
  - Xiaozhu Meng
  - Kevin Roundy
  - Evan Samanas
  - Ben Welton

Summary

- Dyninst Provides
  - Multi Architecture Support (x86, Power)
  - Multi OS Support (Windows, Linux, AIX, VxWorks)
  - Multi Compilter (Intel, Microsoft, GCC, PGI, Cray)
  - Toolkit approach
    - Uses as little or as much as you want
- Dyninst is Mature
  - Commercial Products from IBM & SGI
  - Used in many third party open source tools
- More Information
  - www.dyninst.org