# Modifying Binaries with Dyninst

Andrew Bernat, Bill Williams

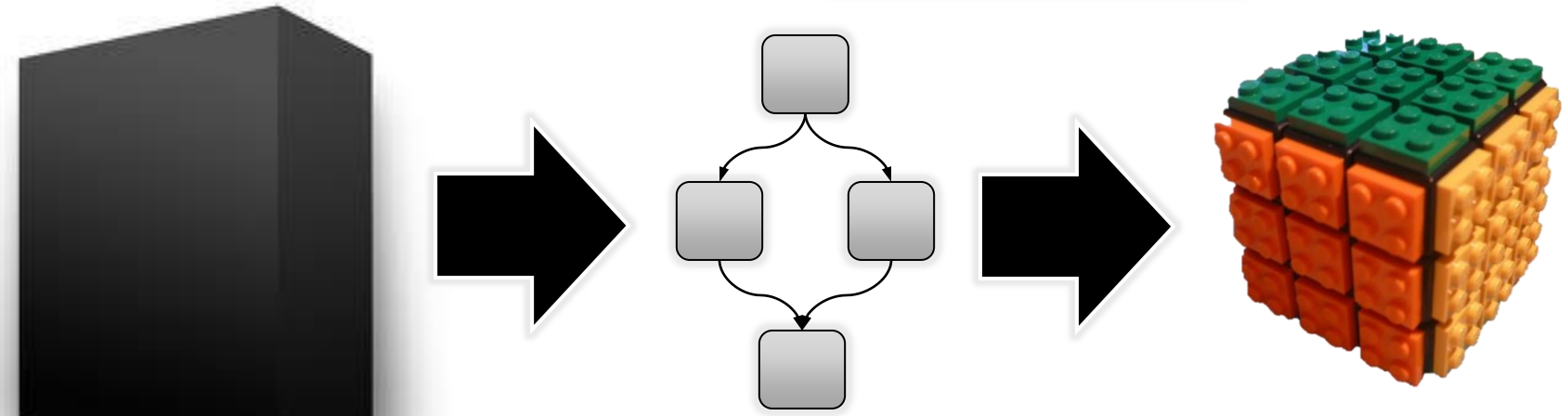
Paradyn Project

Paradyn / Dyninst Week
College Park, Maryland
March 26-28, 2012

# Binary Modification

Predicate switching

Insert error checking and handling



Dynamic patching

Code surgery

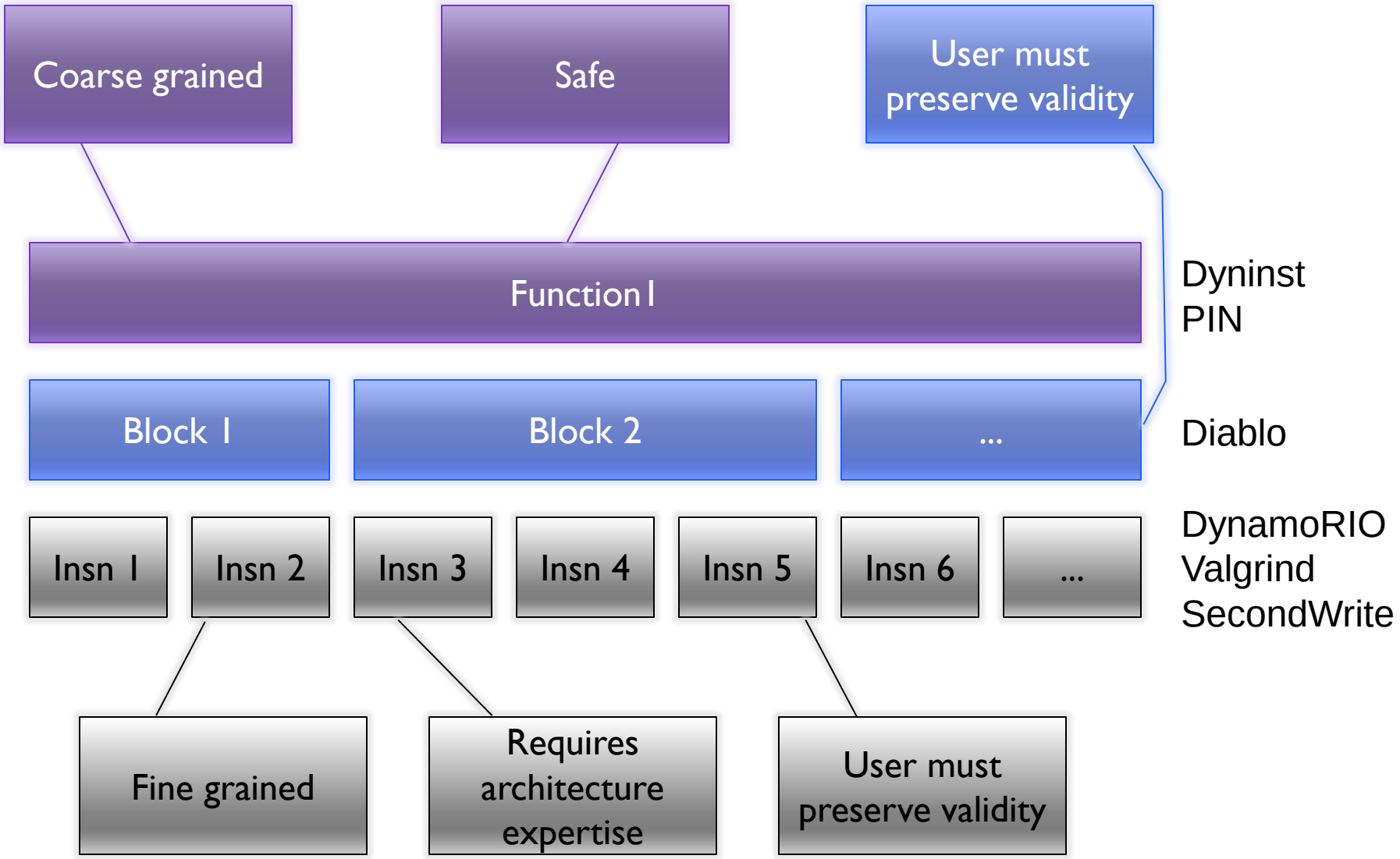
# Goals

- Use familiar abstractions
  - CFG
  - Snippets
- Platform independent
- Safe
  - Avoid side-effects
  - Prevent invalid control flow

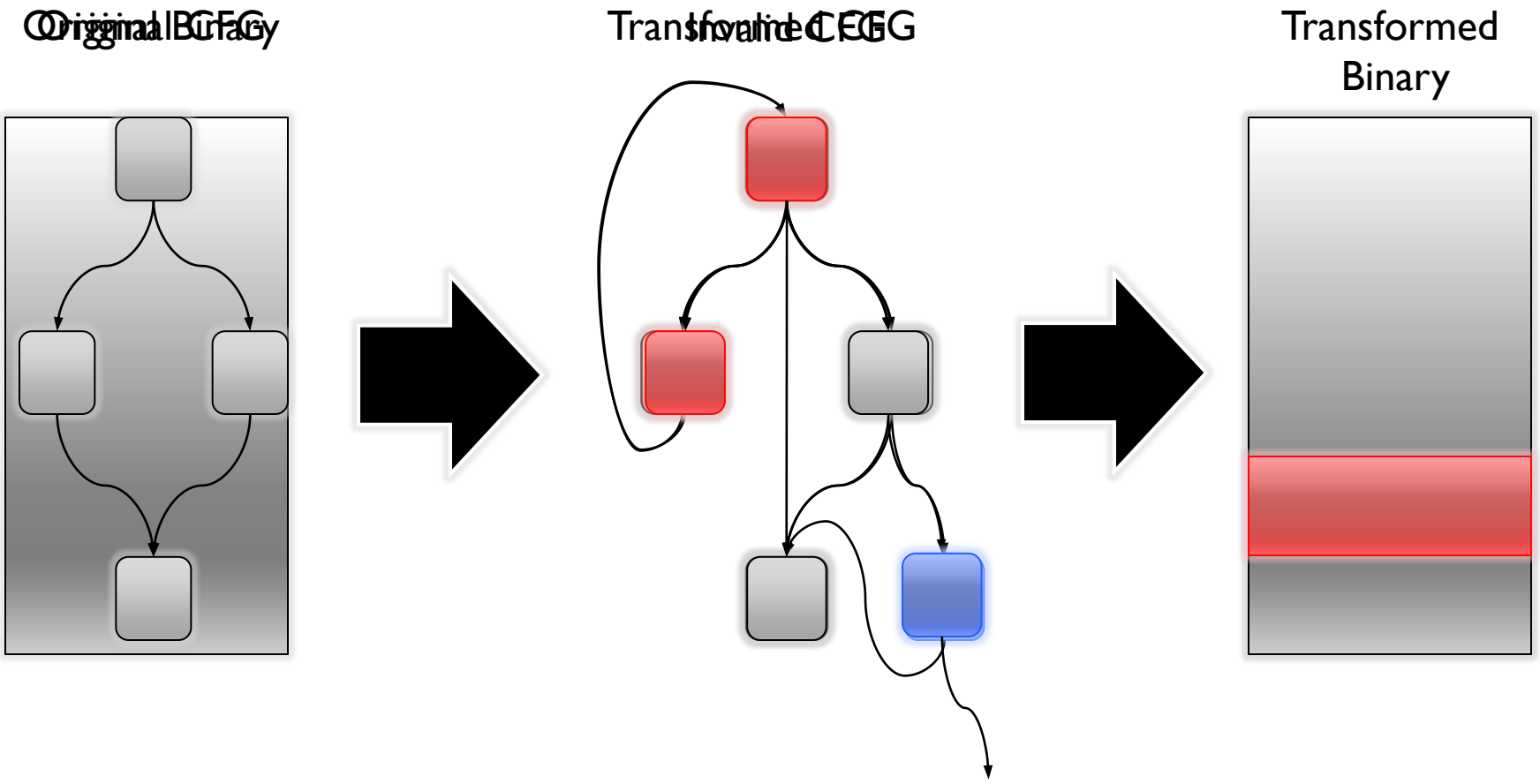
# Outline

- Related Work: Binary Modification Techniques
- Theory: Structured Binary Editing
- Practice: PatchAPI Interface
- Example: Hot Patching Apache Vulnerabilities

# Related Work

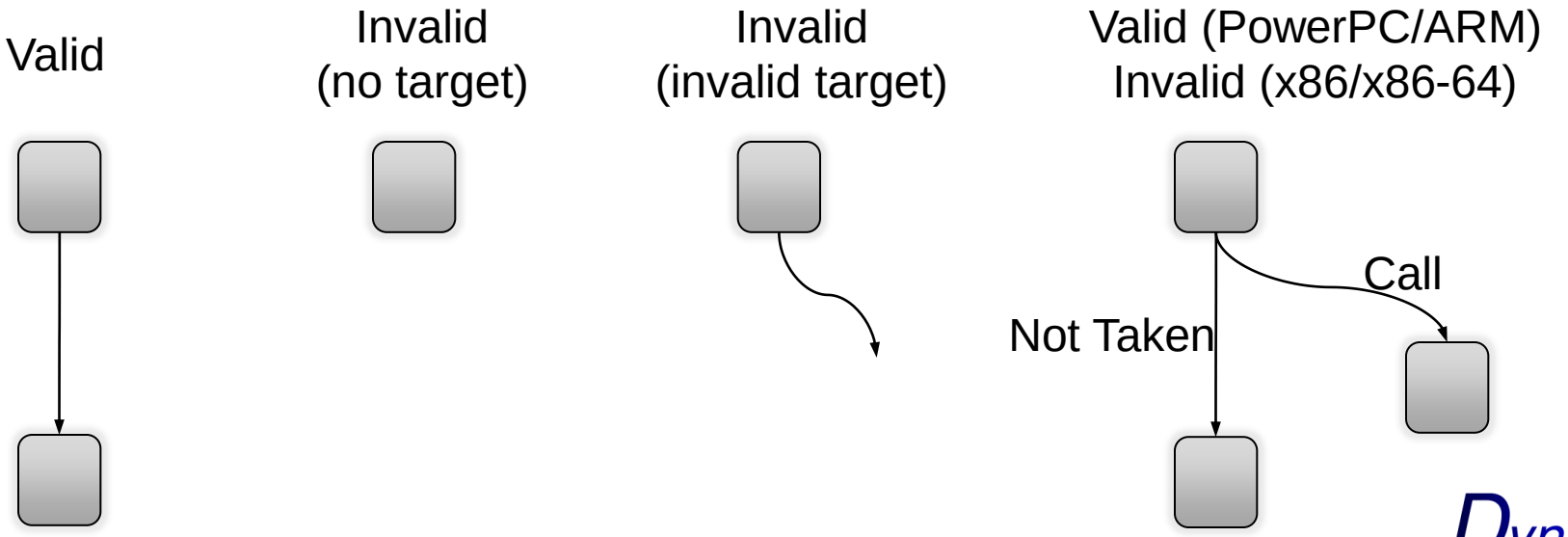


# Theory: Structured Binary Editing



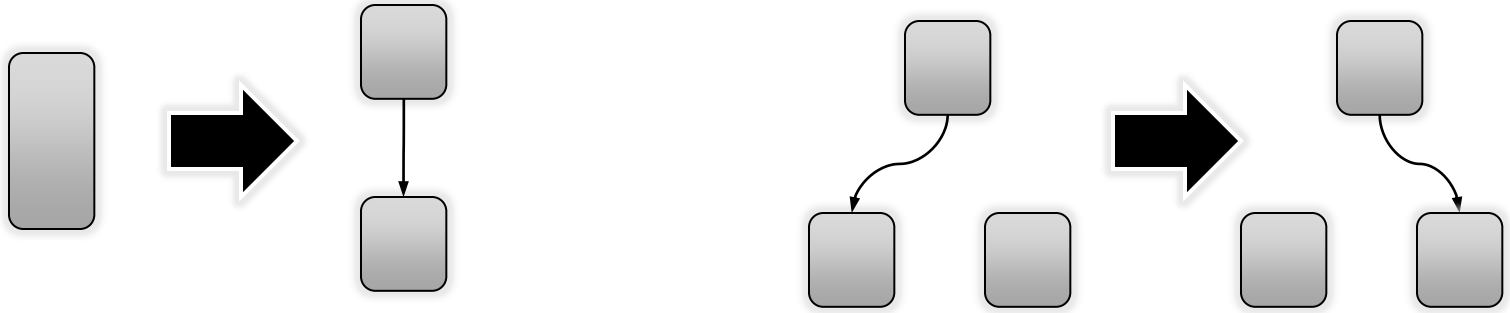
# CFG Validity

- Block validity
  - Blocks contain valid instruction sequences
- Edge validity
  - Edges point to valid (block) targets
  - Edge types correspond with branches/calls

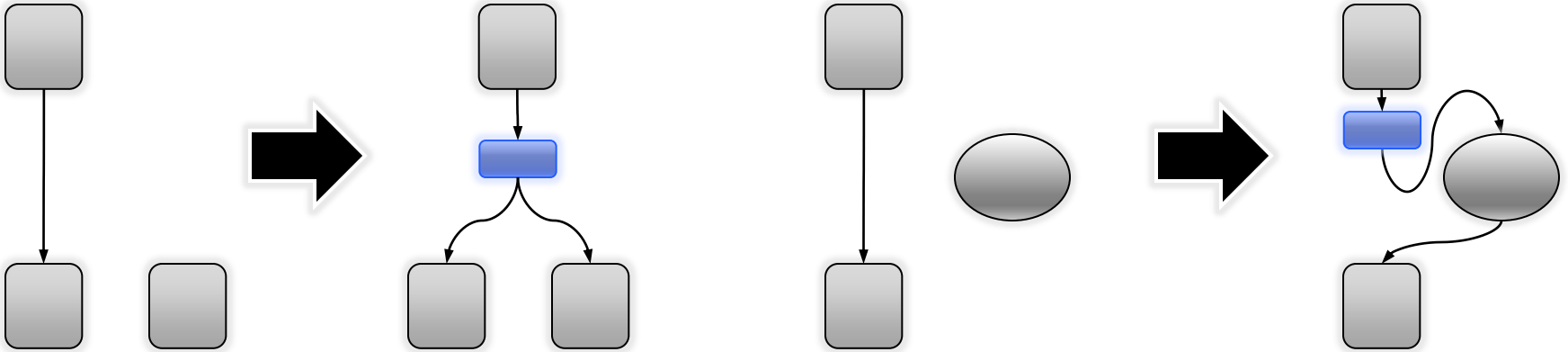


# CFG Transformations

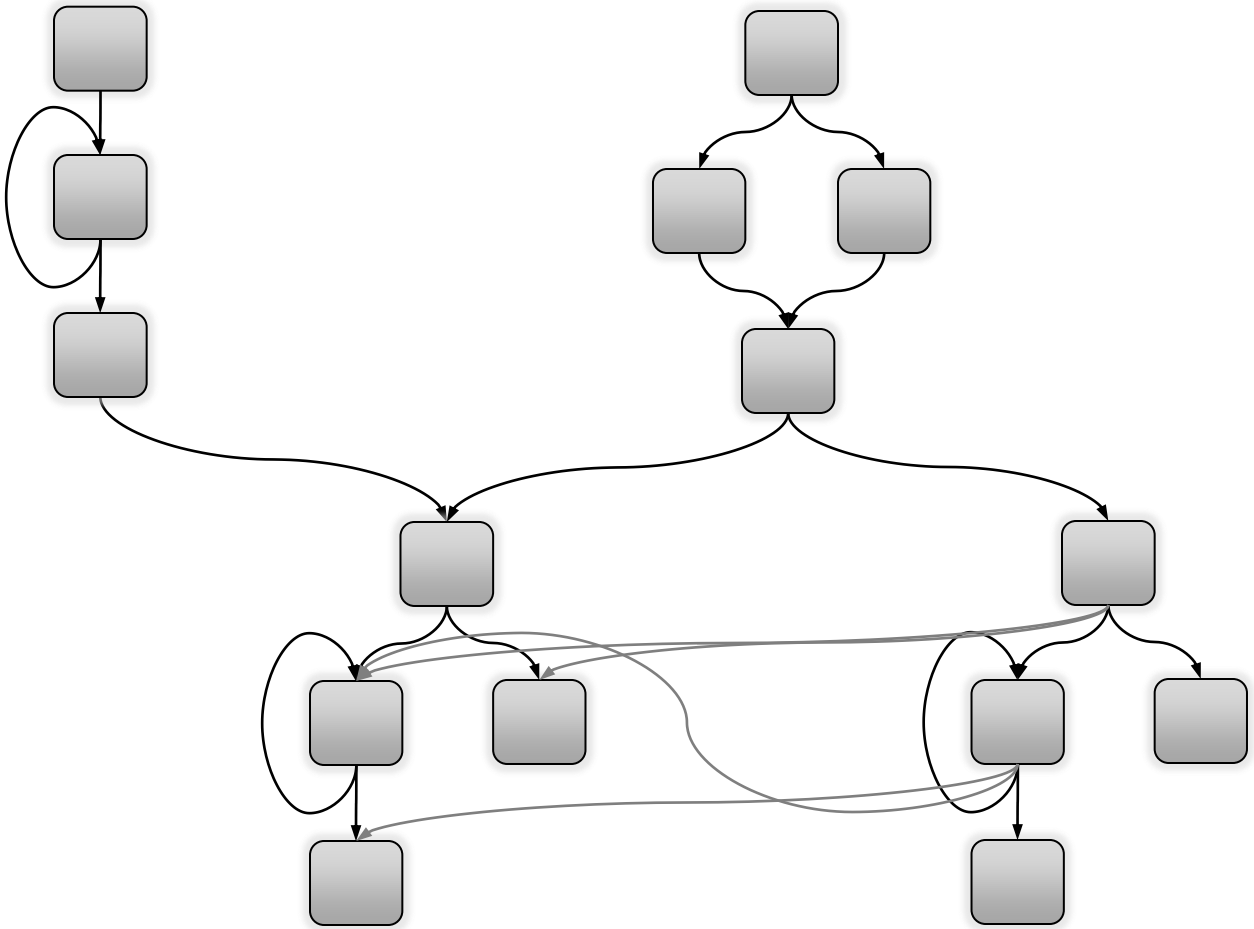
- Simple: block split, edge redirection



- Complex: inserting conditional branches or calls



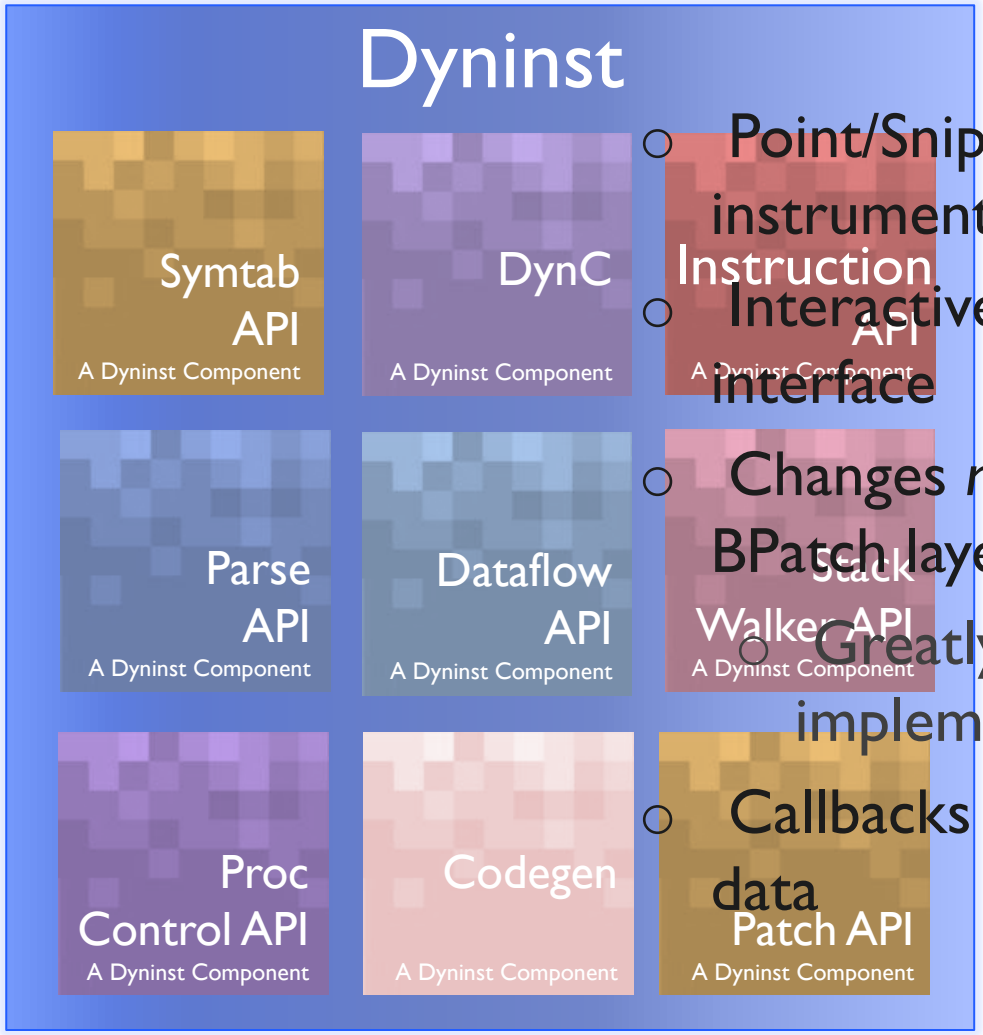
# Composition: Function Unsharing



# Handling Indirect Control Flow

- Transforming the CFG can alter indirect control flow
  - Corrupting function pointer
  - Altering jump table index
- Detect alterations with slice-based analysis
- Spectrum of possible responses:
  - Allow transformation
  - Disallow transformation
  - Insert runtime monitoring code

# Practice: PatchAPI Interface




- Point/Snippet instrumentation interface
- Interactive modification interface
- Changes *not* reflected at BPatch layer
- Greatly simplifies implementation
- Callbacks for updating user data

# PatchAPI Overview

- CFG classes
  - PatchBlock, PatchEdge, PatchFunction
- Binary file class
  - PatchObject
- Instrumentation
  - Point, PatchMgr, Snippet
- Modification
  - PatchModifier, PatchCallback

# (BETA) class Snippet

- bool generate(Point *point, Buffer &buffer)
  - Point provides context for code generation
  - Buffer is a code container
- Convert BPatch_snippets to PatchAPI::Snippets
- Inherit to define your own Snippets

# (BETA) class PatchModifier

- bool redirect(PatchEdge *, PatchBlock *);
- PatchBlock *split(PatchBlock *, Address);
- bool remove(vector<PatchBlock *>);
- bool remove(PatchFunction *);
- InsertedCode::Ptr insert(PatchObject *, SnippetPtr, Point *);
  - class InsertedCode:
    - PatchBlock *entry()
    - vector<PatchEdge *> &exits()
    - set<PatchBlock *> &blocks()

# (BETA) class PatchCallback

- Interface class for CFG modification updates
- Register one (or more) child classes
- Notify on CFG element:
  - Creation
  - Destruction
  - Block splitting
  - New in-edge or out-edge
  - Removed in-edge or out-edge
- Notify on Point creation, destruction, or change

# Example: Hot Patching Apache Vulnerability

- Apache 2.2.21 has several vulnerabilities:
  - Using reverse proxy to access arbitrary server
  - Privilege escalation via .htaccess file
  - Denial of service with badly formatted cookie
  - (and others)
- We used binary modification to patch these flaws
  - Act on unprepared, executing httpd daemon
  - Do not rely on specific compiler version(s)
  - Create "click to run" patching tool

# CVE-2011-3368

- Reverse proxy functionality broken in Apache
  - Introduced in 1.3.x
  - Fixed in 2.3.15 (released November 16, 2011)
- Attacker sends illegal URI
  - GET @hiddenServer/hiddenFile
- Proxy incorrectly forwards request
  - GET proxy@hiddenServer/hiddenFile
- Attacker can access internal files

# Hot Patching Overview

- Create CFG "fingerprint" that identifies patch site(s)
  - Do not rely on addresses, functions, or particular instruction sequences
- Create snippets from security patch file
  - Access and update local variables
  - Insert conditional error handling
- Modify binary to incorporate new code
  - Matching patch file

# Patch File

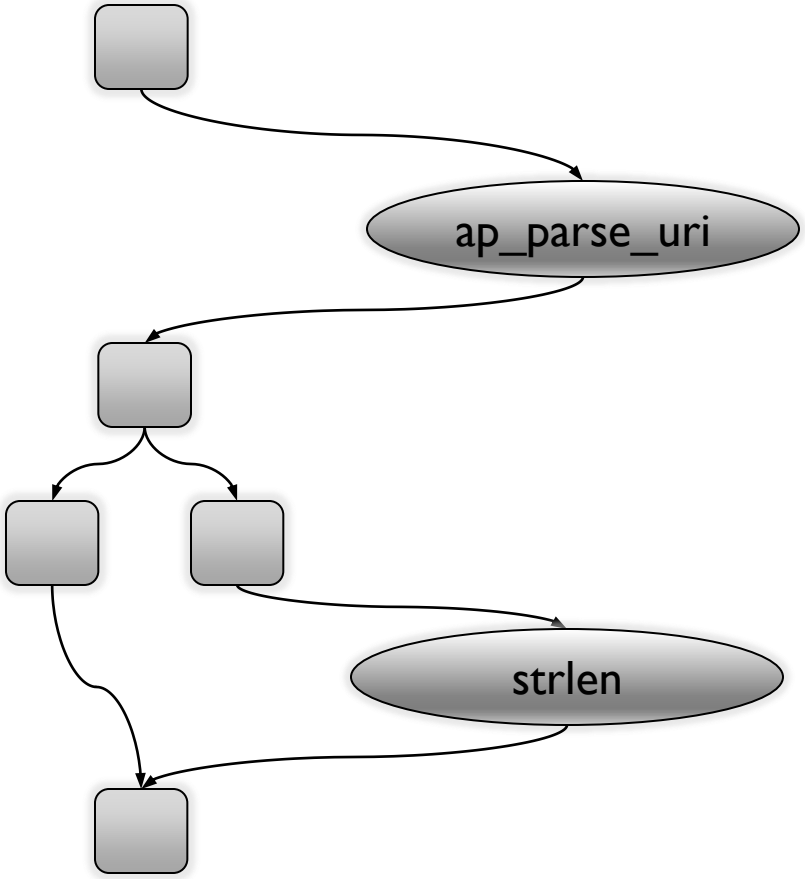
```
  ap_parse_uri(r, uri);

+ if (r->method_number != M_CONNECT
+     && !r->parsed_uri.scheme
+     && uri[0] != '/'
+     && !(uri[0] == '*' && uri[1] == '\0')) {
+   r->args = NULL;
+   r->hostname = NULL;
+   r->status = HTTP_BAD_REQUEST;
+ }

  if (ll[0]) {
    r->assbackwards = 0;
    pro = ll;
  ...
```

Error Condition Detection

Error Handling

# CFG Fingerprinting

```
ap_parse_uri(r, uri);
if (ll[0]) {
  r->assbackwards = 0;
  pro = ll;
  len = strlen(ll);
}
else {
  r->assbackwards = 1;
  pro = “HTTP/0.9”;
  len = 8;
}
```



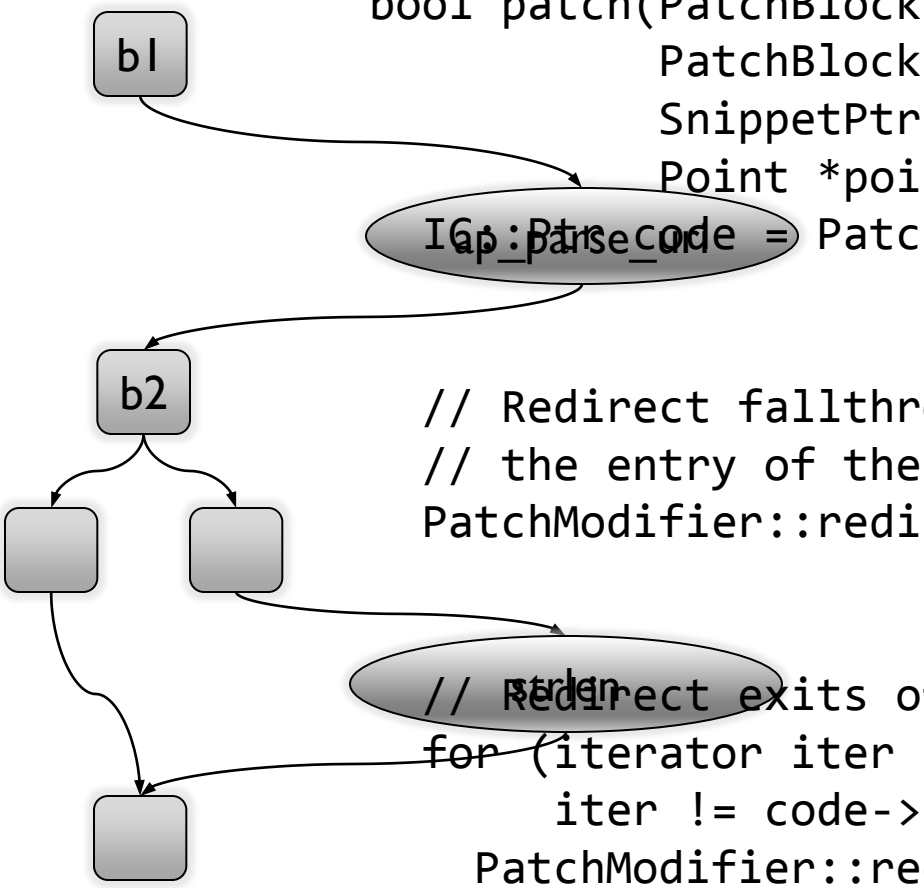
# Snippet (error detection)

```
// r->method_number != M_CONNECT
boolExpr cond1(ne, r_method_number, constExpr(M_CONNECT));

// !r->parsed_uri.scheme
boolExpr cond2(eq, r_parsed_uri_scheme, constExpr(0));

// uri[0] != '/'
boolExpr cond3(ne, arithExpr(deref, uri), constExpr('/'));

// !(uri[0] == '*' && uri[1] == '\0')
boolExpr cond4(or,
               boolExpr(ne, arithExpr(deref, uri), constExpr('*')),
               boolExpr(ne, arithExpr(deref, ...)));

boolExpr cond(and,
              boolExpr(and, cond1, cond2),
              boolExpr(and, cond3, cond4));
```

# Snippet (error handling)

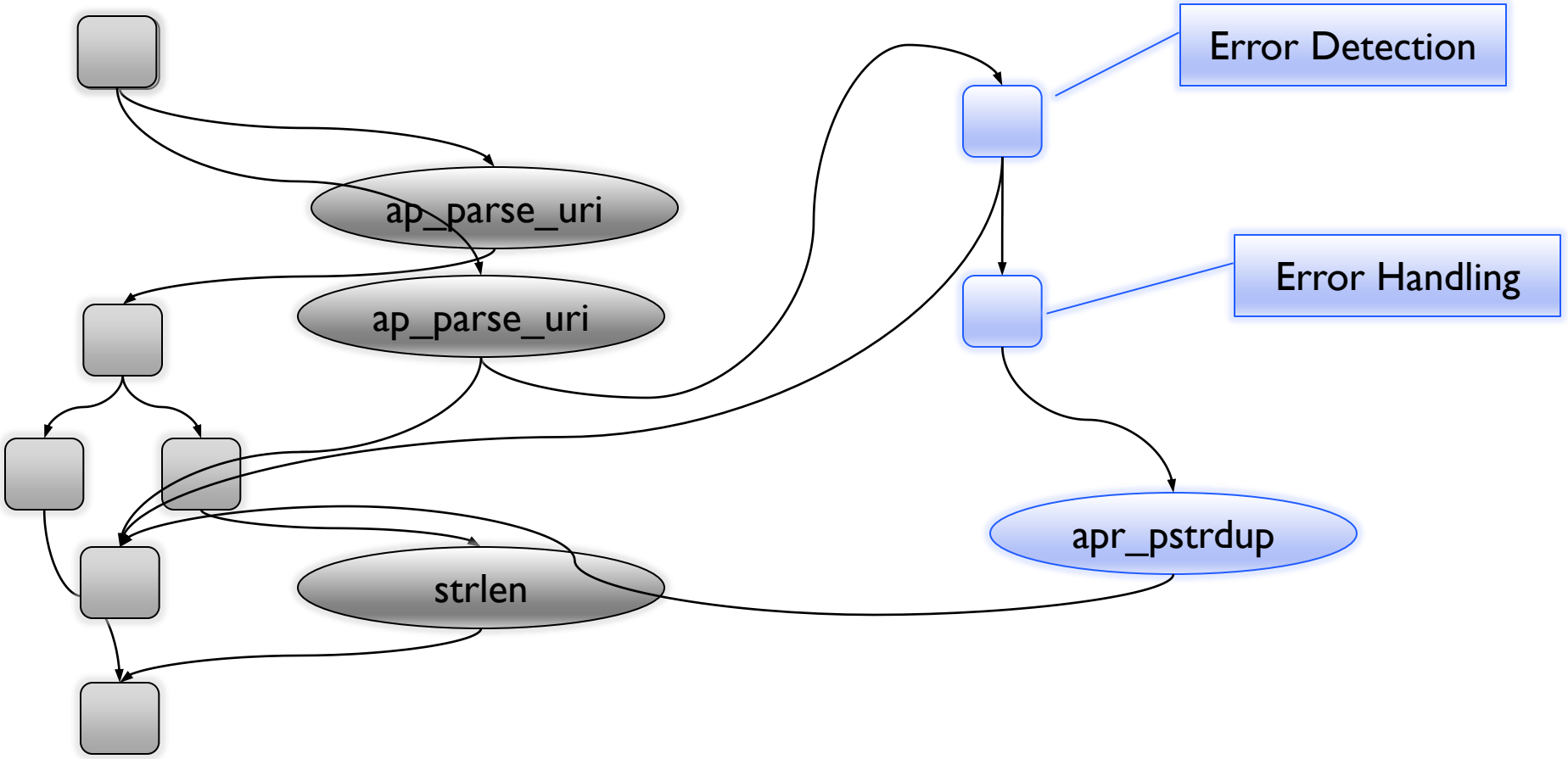
```
// r->args = NULL;
arithExpr body1(assign, r_args, constExpr(0));

// r->hostname = NULL;
arithExpr body2(assign, r_hostname, constExpr(0));

// r->status = HTTP_BAD_REQUEST
arithExpr body3(assign, r_status, constExpr(HTTP_BAD_REQUEST));

// r->uri = apr_pstrdup(r->pool, uri)
vector<snippet> call_args;
call_args.push_back(r_pool);
call_args.push_back(uri);
arithExpr body4(assign, r_uri,
                funcCallExpr(findFunction("apr_pstrdup"),
                             call_args));
```

# Modification Code




```
bool patch(PatchBlock *b1,
           PatchBlock *b2,
           SnippetPtr snip,
           Point *point) {
  IG::Ptr code = PatchModifier::insert(b1->obj(),
                                       snip, point);

  // Redirect fallthrough of ap_parse_uri call to
  // the entry of the inserted code region
  PatchModifier::redirect(getEdge(b1, CALL_FT),
                          code->entry());

  // Redirect exits of inserted code to b2
  for (iterator iter = code->exits().begin();
       iter != code->exits().end(); ++iter) {
    PatchModifier::redirect(*iter, b2);
}
```

# Hot Patching



# Summary

- Structured binary editing
  - Modify binaries by transforming their CFGs
  - Ensure validity of the resulting binary
- PatchAPI implementation
  - Interactive CFG modification
  - Mix modification and instrumentation
  - Callback interface for updating user data

Bernat and Miller,
"Structured Binary Editing with a CFG Transformation Algebra"