Paradyn Parallel Performance Tools

PatchAPI Programmer's Guide

9.3 Release
December 2016

Computer Sciences Department
University of Wisconsin–Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742

Email `dyninst-api@cs.wisc.edu`
Web `www.dyninst.org`

Contents

1 Introduction

This manual describes PatchAPI, a programming interface and library for binary code patching. A programmer uses PatchAPI to instrument (insert code into) and modify a binary executable or library by manipulating the binary's control flow graph (CFG). We allow the user to instrument a binary by annotating a CFG with *snippets*, or sequences of inserted code, and to modify the binary by directly manipulating the CFG. The PatchAPI interface, and thus tools written with PatchAPI, is designed to be flexible and extensible. First, users may *inherit* from PatchAPI abstractions in order to store their own data. Second, users may create *plugins* to extend PatchAPI to handle new types of instrumentation, different binary types, or different patching techniques.

PatchAPI represents the binary as an annotatable and modifiable CFG. The CFG consists of abstractions for binary objects, functions, basic blocks, edges connecting basic blocks, and loops representing code that may execute repeatedly, which are similar to the CFG abstractions used by the ParseAPI component.

Users instrument the binary by annotating this CFG using three additional high-level abstractions: Point, Snippet, and Instance. A Point supports instrumentation by representing a particular aspect of program behavior (e.g., entering a function or traversing an edge) and containing instances of Snippets. Point lookup is performed with a single PatchAPI manager (PatchMgr) object by Scope (e.g., a CFG object) and Type (e.g., function entry). In addition, a user may provide an optional Filter that selects a subset of matching Points. A Snippet represents a sequence of code to be inserted at certain points. To maximize flexibility, PatchAPI does not prescribe a particular snippet form; instead, users may provide their own (e.g., a binary buffer, a Dyninst abstract syntax tree (AST), or code written in the DynC language). Users instrument the binary by adding Snippets to the desired Points. An Instance represents the insertion of a particular Snippet at a particular Point.

The core PatchAPI representation of an annotatable and modifiable CFG operates in several domains, including on a running process (dynamic instrumentation) or a file on disk (binary rewriting). Furthermore, PatchAPI may be used both in the same address space as the process (1st-party instrumentation) or in a different address space via the debug interface (3rd-party instrumentation). Similarly, developers may define their own types of Snippets to encapsulate their own code generation techniques. These capabilities are provided by a plugin interface; by implementing a plugin a developer may extend PatchAPI's capabilities.

This manual is structured as follows. Section 2 presents the core abstractions in the public and plugin interface of PatchAPI. Section 3 shows several examples in C++ to illustrate the usage of PatchAPI. Detailed API reference can be found in Section 4 and Section 6. Finally, Appendix A provides a quick tutorial of PatchAPI to those who are already familiar with DyninstAPI.

2 Abstractions

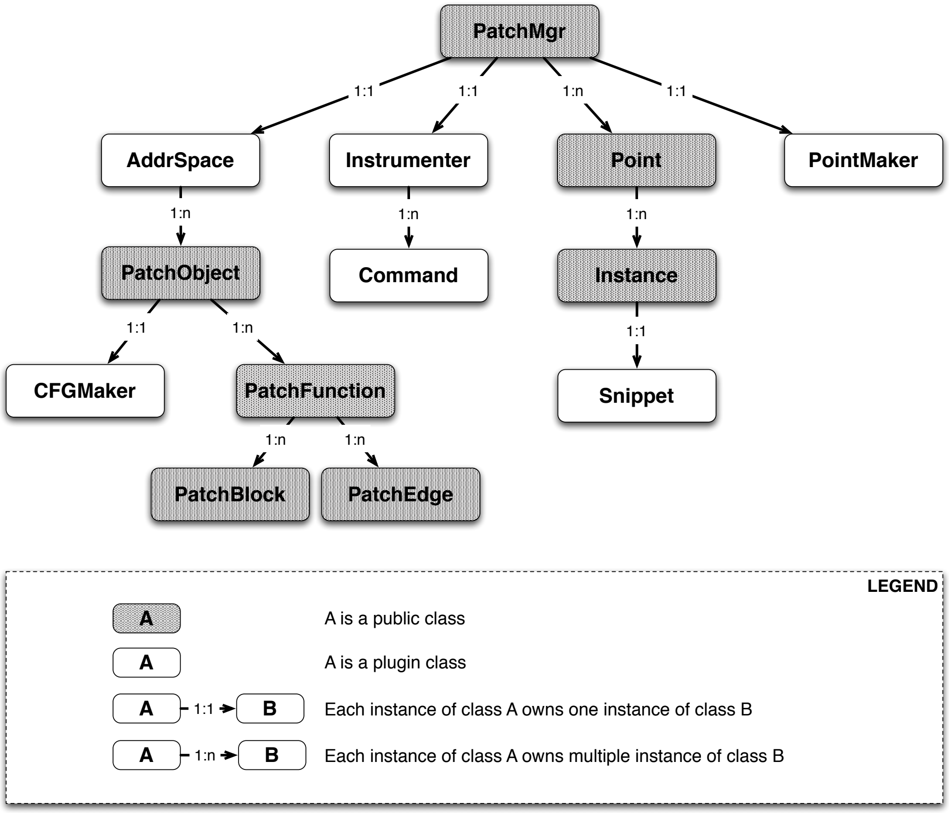

Figure 1: Object Ownership

PatchAPI contains two interfaces: the public interface and the plugin interface. The public interface is used to find instrumentation points, insert or delete code snippets, and register plugins provided by programmers. The plugin interface is used to customize different aspects in the binary code patching. PatchAPI provides a set of default plugins for first party code patching, which is easy to extend to meet different requirements in practice.

Figure 1 shows the ownership hierarchy for PatchAPI's classes. Ownership is a "contains" relationship. If one class owns another, then instances of the owner class maintain exactly one or possibly more than one instances of the other, which depends on whether the relationship is a "1:1" or a "1:n" relationship. In Figure 1, for example, each PatchMgr instance contains exactly one instance of a AddrSpace object, while a PatchMgr instance may contains more than one instances of a Point object.

The remainder of this section briefly describes the classes that make up PatchAPI's two interfaces. For more details, see the class descriptions in Section 4 and Section 6.

2.1 Public Interface

PatchMgr, Point, and Snippet are used to perform the main process of binary code patching: 1) find some **Point**; 2) insert or delete **Snippet** at some **Point**.

- *PatchMgr* - The PatchMgr class is the top-level class for finding instrumentation **Points**, inserting or deleting **Snippets**, and registering user-provided plugins.
- *Point* - The Point class represents a location on the CFG that acts as a container of inserted snippet **Instances**. Points of different types are distinct even the underlying code relocation and generation engine happens to put instrumentation from them at the same place.
- *Instance* - The Instance class is a representation of a particular snippet inserted at a particular point.
- *PatchObject* - The PatchObject class is a wrapper of ParseAPI's CodeObject class, which represents an individual binary code object, such as an executable or a library.
- *PatchFunction* - The PatchFunction class is a wrapper of ParseAPI's Function class, which represents a function.
- *PatchBlock* - The PatchBlock class is a wrapper of ParseAPI's Block class, which represents a basic block.
- *PatchEdge* - The PatchEdge class is a wrapper of ParseAPI's Edge class, which join two basic blocks in the CFG, indicating the type of control flow transfer instruction that joins the basic blocks to each other.
- *PatchLoop* - The PatchLoop class is a wrapper of ParseAPI's Loop class, which repreents a piece of code that may execute repeatedly.
- *PatchLoopTreeNode* - The PatchLoopTreeNode class is a wrapper of ParseAPI's LoopTreeNode class, which provides a tree interface to a collection of instances of class PatchLoop contained in a function. The structure of the tree follows the nesting relationship of the loops in a function.

2.2 Plugin Interface

The address space abstraction determines whether the code patching is 1st party, 3rd party or binary rewriting.

- *AddrSpace* - The AddrSpace class represents the address space of a **Mutatee** (a program that is instrumented), where it contains a collection of **PatchObjects** that represent shared libraries or a binary executable. In addition, programmers implement some memory management interfaces in the AddrSpace class to determines the type of the code patching - 1st party, 3rd party, or binary rewriting.

Programmers can decide the representation of a **Snippet**, for example, the representation can be in high level language (e.g., C or C++), or can simply be in binary code (e.g., 0s and 1s).

- *Snippet* - The Snippet class allows programmers to easily plug in their own snippet representation and the corresponding mini-compiler to translate the representation into the binary code.

PatchAPI provides a thin layer on top of ParseAPI's Control Flow Graph (CFG) layer, which associates some useful information for the ease of binary code patching, for example, a shared library's load address. This layer of CFG structures include PatchObject, PatchFunction, PatchBlock and PatchEdge classes. Programmers can extend these four CFG classes, and use the derived class of CFGMaker to build a CFG with the augmented CFG structures.

- *CFGMaker* - The CFGMaker class is a factory class that constructs the above CFG structures. This class is used in CFG parsing.

Similar to customizing the PatchAPI layer, programmers can also customize the Point class by extending it.

- *PointMaker* - The PointMaker class is a factory class that constructs a subclass of the Point class.

The implementation of an instrumentation engine may be very sophisticated (e.g., relocating a function), or very simple (e.g., simply overwrite an instruction). Therefore, PatchAPI provides a flexible framework for programmers to customize the instrumentation engine.

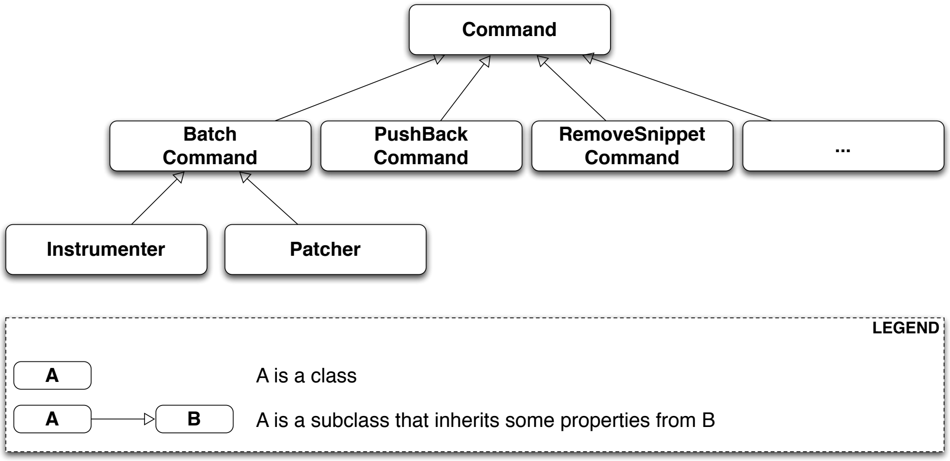

Figure 2: Inheritance Hierarchy

This framework is based on Command Pattern [1]. The instrumentation engine has transactional semantics, where all instrumentation requests should succeed or all should fail. In our framework, the **Command** abstraction represents an instrumentation request or a logical step in the code patching process. We accumulate a list of **Commands**, and execute them one by one. If one **Command** fails, we undo all preceding finished **Commands**. Figure 2 illustrates the inheritance hierarchy for related classes. There is a default implementation of instrumentation engine in PatchAPI for 1st party code patching.

- *Command* - The Command class represents an instrumentation request (e.g., snippet insertion or removal), or a logical step in the code patching (e.g., install instrumentation). This class provides a run() method and an undo() method, where run() will be called for normal execution, and undo() will be called for undoing this Command.
- *BatchCommand* - The BatchCommand class is a subclass of Command, and it is in fact a container of a list of Commands to be executed atomically.
- *Instrumenter* - The Instrumenter class inherits BatchCommand to encapsulate the core code patching logic, which includes binary code generation. Instrumenter would contain several logical steps that are individual Commands.
- *Patcher* - The Patcher class is also a subclass of BatchCommand. It accepts instrumentation requests from users, where these instrumentation requests are Commands (e.g., snippet insertion). Furthermore, Patcher implicitly adds Instrumenter to the end of the Command list to generate binary code and install the instrumentation.

---

[1] http://en.wikipedia.org/wiki/Command_pattern

3 Examples

To illustrate the ideas of PatchAPI, we present some simple code examples that demonstrate how the API can be used.

3.1 Using the public interface

The basic flow of doing code patching is to first find some points in a program, and then to insert, delete or update a piece of code at these points.

3.1.1 CFG Traversal

Listing 1: Example of CFG traversal

```
ParseAPI::CodeObject* co = ...
PatchObject* obj = PatchObject::create(co, code_base);

// Find all functions in the object
std::vector<PatchFunction*> all;
obj->funcs(back_inserter(all));

for (std::vector<PatchFunction*>::iterator fi = all.begin();
     fi != all.end(); fi++) {
  // Print out each function's name
  PatchFunction* func = *fi;
  std::cout << func->name() << std::endl;

  const PatchFunction::Blockset& blks = func->blocks();
  for (PatchFunction::BlockSet::iterator bi = blks.begin();
       bi != blks.end(); bi++) {
    // Print out each block's size
    PatchBlock* blk = *bi;
    std::cout << "\tBlock size:" << blk->size() << std::endl;
  }
}
```

In the above code, we illustrate how to traverse CFG structures in PatchAPI. First, we construct an instance of PatchObject using an instance of ParseAPI's CodeObject. Then, we traverse all functions in that object, and print out each function's name. For each function, we also print out the size of each basic block.

3.1.2 Point Finding

Listing 2: Example of point finding

```
1  PatchFunction *func = ...;
2  PatchBlock *block = ...;
3  PatchEdge *edge = ...;
4
5  PatchMgr *mgr = ...;
6
7  std::vector<Point*> pts;
8  mgr->findPoints(Scope(func),
9                  Point::FuncEntry |
10                 Point::PreCall |
11                 Point::FuncExit,
12                 back_inserter(pts));
13 mgr->findPoints(Scope(block),
14                 Point::BlockEntry,
15                 back_inserter(pts));
16 mgr->findPoints(Scope(edge),
17                 Point::EdgeDuring,
18                 back_inserter(pts));
```

The above code shows how to use the PatchMgr::findPoints method to find some instrumentation points. There are three invocations of findPoints. For the first invocation (Line 8), it finds points only within a specific function *func*, and output the found points to a vector *pts*. The result should include all points at this function's entry, before all function calls inside this function, and at the function's exit. Similarly, for the second invocation (Line 13), it finds points only within a specific basic *block*, and the result should include the point at the block entry. Finally, for the third invocation (Line 16), it finds the point at a specific CFG *edge* that connects two basic blocks.

3.1.3 Code Patching

Listing 3: Example of code patching

```
1  MySnippet::ptr snippet = MySnippet::create(new MySnippet);
2
3  Patcher patcher(mgr);
4  for (vector<Point*>::iterator iter = pts.begin();
5       iter != pts.end(); ++iter) {
6    Point* pt = *iter;
7    patcher.add(PushBackCommand::create(pt, snippet));
8  }
9  patcher.commit();
```

The above code is to insert the same code *snippet* to all points *pts* found in Section 3.1.2. We'll explain the snippet (Line 1) in the example in Section 3.2.2. Each point maintains a list of snippet instances, and the PushBackCommand is to push a snippet instance to the end of that list. An instance of Patcher is to represent a transaction of code patching. In this example, all snippet insertions (or all PushBackCommands) are performed atomically when the Patcher::commit method is invoked. That is, all snippet insertions would succeed or all would fail.

3.2 Using the plugin interface

3.2.1 Address Space

Listing 4: Example of implementing address space plugin

```
class MyAddrSpace : public AddrSpace {
  public:
    ...
    virtual Address malloc(PatchObject* obj, size_t size, Address near) {
      Address buffer = ...
      // do memory allocation here
      return buffer;
    }
    virtual bool write(PatchObject* obj, Address to_addr, Address from_addr,
                       size_t size) {
      // copy data from the address from_addr to the address to_addr
      return true;
    }
    ...
};
```

The above code is to implement the address space plugin, in which, a set of memory management methods should be specified, including malloc, free, realloc, write and so forth. The instrumentation engine will utilize these memory management methods during the code patching process. For example, the instrumentation engine needs to *malloc* a buffer in Mutatee's address space, and then *write* the code snippet into this buffer.

3.2.2 Snippet Representation

Listing 5: Example of implementing snippet plugin

```
class MySnippet : public Snippet {
  public:
    virtual bool generate(Point *pt, Buffer &buf) {
      // Generate and store binary code in the Buffer buf
      return true;
    }
};
MySnippet::ptr snippet = MySnippet::create(new MySnippet);
```

The above code illustrates how to customize a user-defined snippet *MySnippet* by implementing the "mini-compiler" in the *generate* method, which will be used later in the instrumentation engine to generate binary code.

3.2.3 Code Parsing

Listing 6: Example of customizing CFG parsing

```
class MyFunction : public PatchFunction {
  ...
};
class MyCFGMaker : public CFGMaker {
  public:
    ...
    virtual PatchFunction* makeFunction(ParseAPI::Function *f, PatchObject* o) {
      return new MyFunction(f, o);
    }
    ...
};
```

Programmers can augment PatchAPI's CFG structures by annotating their own data. In this case, a factory class should be built by inheriting from the CFGMaker class, to create the augmented CFG structures. The factory class will be used for CFG parsing.

3.2.4 Point Making

Listing 7: Example of point making

```
class MyPoint : public Point {
  public:
    MyPoint(Point::Type t, PatchMgrPtr m, PatchFunction *f);
    ...
};

class MyPointMaker: public PointMaker {
  protected:
    virtual Point *mkFuncPoint(Point::Type t, PatchMgrPtr m, PatchFunction *f) {
      return new MyPoint(t, m, f);
    }
};
```

In the above example, the MyPoint class inherits from the Point class, and the MyPointMaker class inherits from the PointMaker class. The mkFuncPoint method in MyPointMaker simply returns a new instance of MyPoint. The mkFuncPoint method will be invoked by PatchMgr::findPoint(s).

3.2.5 Instrumentation Engine

Listing 8: Example of customizing instrumentation engine

```
class MyInstrumenter : public Instrumenter {
  public:
    virtual bool run() {
      // Specify how to install instrumentation
    }
};
```

Programmers can customize the instrumentation engine by extending the Instrumenter class, and implement the installation of instrumentation inside the method *run()*.

3.2.6 Plugin Registration

Listing 9: Example of registering plugins

```
 1 MyCFGMakerPtr cm = ...
 2 PatchObject* obj = PatchObject::create(..., cm);
 3
 4 MyAddrSpacePtr as = ...
 5 as->loadObject(obj);
 6
 7 MyInstrumenter inst = ...
 8 PatchMgrPtr mgr = PatchMgr::create(as, ..., inst);
 9
10 MySnippet::ptr snippet = MySnippet::create(new MySnippet);
```

The above code shows how to register the above four types of plugins. An instance of the factory class for creating CFG structures is registered to an PatchObject (Line 1 and 2), which is in turn loaded into an instance of AddrSpace (Line 4 and 5). The AddrSpace (or its subclass implemented by programmers) instance is passed to PatchMgr::create (Line 7 and 8), together with an instance of Instrumenter (or its subclass). Finally, a snippet of custom snippet representation MySnippet is created (Line 10). Therefore, all plugins are glued together in PatchAPI.

4 Public API Reference

This section describes public interfaces in PatchAPI. The API is organized as a collection of C++ classes. The classes in PatchAPI fall under the C++ namespace Dyninst::PatchAPI. To access them, programmers should refer to them using the "Dyninst::PatchAPI::" prefix, e.g., Dyninst::PatchAPI::Point. Alternatively, programmers can add the C++ *using* keyword above any references to PatchAPI objects, e.g.,*using namespace Dyninst::PatchAPI* or *using Dyninst::PatchAPI::Point.*

Classes in PatchAPI use either the C++ raw pointer or the boost shared pointer (*boost::shared_ptr<T>*) for memory management. A class uses a raw pointer whenever it is returning a handle to the user that is controlled and destroyed by the PatchAPI runtime library. Classes that use a raw pointer include the CFG objects, a Point, and various plugins, e.g., AddrSpace, CFGMaker, PointMaker, and Instrumenter. A class uses a shared_pointer whenever it is handing something to the user that the PatchAPI runtime library is not controlling and destroying. Classes that use a boost shared pointer include a Snippet, PatchMgr, and Instance, where we typedef a class's shared pointer by appending the Ptr to the class name, e.g., PatchMgrPtr for PatchMgr.

4.1 CFG Interface

4.1.1 PatchObject

Declared in: PatchObject.h

The PatchObject class is a wrapper of ParseAPI's CodeObject class (has-a), which represents an individual binary code object, such as an executable or a library.

```
static PatchObject* create(ParseAPI::CodeObject* co, Address base,
                           CFGMaker* cm = NULL, PatchCallback *cb = NULL);
```

Creates an instance of PatchObject, which has *co* as its on-disk representation (ParseAPI::CodeObject), and *base* as the base address where this object is loaded in the memory. For binary rewriting, base should be 0. The *cm* and *cb* parameters are for registering plugins. If *cm* or *cb* is NULL, then we use the default implementation of CFGMaker or PatchCallback.

```
static PatchObject* clone(PatchObject* par_obj, Address base,
                          CFGMaker* cm = NULL, PatchCallback *cb = NULL);
```

Returns a new object that is copied from the specified object *par_obj* at the loaded address *base* in the memory. For binary rewriting, base should be 0. The *cm* and *cb* parameters are for registering plugins. If *cm* or *cb* is NULL, then we use the default implementation of CFGMaker or PatchCallback.

```
Address codeBase();
```

Returns the base address where this object is loaded in memory.

```
PatchFunction *getFunc(ParseAPI::Function *func, bool create = true);
```

Returns an instance of PatchFunction in this object, based on the *func* parameter. PatchAPI creates a PatchFunction on-demand, so if there is not any PatchFunction created for the ParseAPI function *func*, and the *create* parameter is false, then no any instance of PatchFunction will be created.

It returns NULL in two cases. First, the function *func* is not in this PatchObject. Second, the PatchFunction is not yet created and the *create* is false. Otherwise, it returns a PatchFunction.

```
template <class Iter>
void funcs(Iter iter);
```

Outputs all instances of PatchFunction in this PatchObject to the STL inserter *iter*.

```
PatchBlock *getBlock(ParseAPI::Block* blk, bool create = true);
```

Returns an instance of PatchBlock in this object, based on the *blk* parameter. PatchAPI creates a PatchBlock on-demand, so if there is not any PatchBlock created for the ParseAPI block *blk*, and the *create* parameter is false, then no any instance of PatchBlock will be created.

It returns NULL in two cases. First, the ParseAPI block *blk* is not in this PatchObject. Second, the PatchBlock is not yet created and the *create* is false. Otherwise, it returns a PatchBlock.

```
template <class Iter>
void blocks(Iter iter);
```

Outputs all instances of PatchBlock in this object to the STL inserter *iter*.

```
PatchEdge *getEdge(ParseAPI::Edge* edge, PatchBlock* src, PatchBlock* trg,
                   bool create = true);
```

Returns an instance of PatchEdge in this object, according to the parameters ParseAPI::Edge *edge*, source PatchBlock *src*, and target PatchBlock *trg*. PatchAPI creates a PatchEdge on-demand, so if there is not any PatchEdge created for the ParseAPI *edge*, and the *create* parameter is false, then no any instance of PatchEdge will be created.

It returns NULL in two cases. First, the ParseAPI *edge* is not in this PatchObject. Second, the PatchEdge is not yet created and the *create* is false. Otherwise, it returns a PatchEdge.

```
template <class Iter>
void edges(Iter iter);
```

Outputs all instances of PatchEdge in this object to the STL inserter *iter*.

```
PatchCallback *cb() const;
```

Returns the PatchCallback object associated with this PatchObject.

4.1.2 PatchFunction

Declared in: PatchCFG.h

The PatchFunction class is a wrapper of ParseAPI's Function class (has-a), which represents a function.

```
const string &name();
```

Returns the function's mangled name.

`Address addr() const;`

Returns the address of the first instruction in this function.

`ParseAPI::Function *function();`

Returns the ParseAPI::Function associated with this PatchFunction.

`PatchObject* obj();`

Returns the PatchObject associated with this PatchFunction.

`typedef std::set<PatchBlock *> PatchFunction::Blockset;`

`const Blockset &blocks();`

Returns a set of all PatchBlocks in this PatchFunction.

`PatchBlock *entry();`

Returns the entry block of this PatchFunction.

`const Blockset &exitBlocks();`

Returns a set of exit blocks of this PatchFunction.

`const Blockset &callBlocks();`

Returns a set of all call blocks of this PatchFunction.

`PatchCallback *cb() const;`

Returns the PatchCallback object associated with this PatchFunction.

`PatchLoopTreeNode* getLoopTree()`

Return the nesting tree of the loops in the function. See class `PatchLoopTreeNode` for more details

`PatchLoop* findLoop(const char *name)`

Return the loop with the given nesting name. See class `PatchLoopTreeNode` for more details about how loop nesting names are assigned.

`bool getLoops(vector<PatchLoop*> &loops);`

Fill `loops` with all the loops in the function

`bool getOuterLoops(vector<PatchLoop*> &loops);`

Fill `loops` with all the outermost loops in the function

`bool dominates(PatchBlock* A, PatchBlock *B);`

Return true if block `A` dominates block `B`

`PatchBlock* getImmediateDominator(PatchBlock *A);`

Return the immediate dominator of block `A`ïijŇ`NULL` if the block `A` does not have an immediate dominator.

`void getImmediateDominates(PatchBlock *A, set<PatchBlock*> &imm);`

Fill `imm` with all the blocks immediate dominated by block `A`

`void getAllDominates(PatchBlock *A, set<PatchBlock*> &dom);`

Fill `dom` with all the blocks dominated by block `A`

```
bool postDominates(PatchBlock* A, PatchBlock *B);
```

Return true if block `A` post-dominates block `B`

```
PatchBlock* getImmediatePostDominator(PatchBlock *A);
```

Return the immediate post-dominator of block `A`ïijŇ`NULL` if the block `A` does not have an immediate post-dominator.

```
void getImmediatePostDominates(PatchBlock *A, set<PatchBlock*> &imm);
```

Fill `imm` with all the blocks immediate post-dominated by block `A`

```
void getAllPostDominates(PatchBlock *A, set<PatchBlock*> &dom);
```

Fill `dom` with all the blocks post-dominated by block `A`

4.1.3 PatchBlock

Declared in: PatchCFG.h

The PatchBlock class is a wrapper of ParseAPI's Block class (has-a), which represents a basic block.

```
Address start() const;
```

Returns the lower bound of this block (the address of the first instruction).

```
Address end() const;
```

Returns the upper bound (open) of this block (the address immediately following the last byte in the last instruction).

```
Address last() const;
```

Returns the address of the last instruction in this block.

```
Address size() const;
```

Returns end() - start().

```
bool isShared();
```

Indicates whether this block is contained by multiple functions.

```
int containingFuncs() const;
```

Returns the number of functions that contain this block.

```
typedef std::map<Address, InstructionAPI::Instruction::Ptr> Insns;
void getInsns(Insns &insns) const;
```

This function outputs Instructions that are in this block to *insns*.

```
InstructionAPI::Instruction::Ptr getInsn(Address a) const;
```

Returns an Instruction that has the address a as its starting address. If no any instruction can be found in this block with the starting address a, it returns InstructionAPI::Instruction::Ptr().

```
std::string disassemble() const;
```

Returns a string containing the disassembled code for this block. This is mainly for debugging purpose.

```
bool containsCall();
```

Indicates whether this PatchBlock contains a function call instruction.

`bool containsDynamicCall();`

> Indicates whether this PatchBlock contains any indirect function call, e.g., via function pointer.

`PatchFunction* getCallee();`

> Returns the callee function, if this PatchBlock contains a function call; otherwise, NULL is returned.

`PatchFunction *function() const;`

> Returns a PatchFunction that contains this PatchBlock. If there are multiple PatchFunctions containing this PatchBlock, then a random one of them is returned.

`ParseAPI::Block *block() const;`

> Returns the ParseAPI::Block associated with this PatchBlock.

`PatchObject* obj() const;`

> Returns the PatchObject that contains this block.

`typedef std::vector<PatchEdge*> PatchBlock::edgelist;`

`const edgelist &sources();`

> Returns a list of the source PatchEdges. This PatchBlock is the target block of the returned edges.

`const edgelist &targets();`

> Returns a list of the target PatchEdges. This PatchBlock is the source block of the returned edges.

```
template <class OutputIterator>
void getFuncs(OutputIterator result);
```

Outputs all functions containing this PatchBlock to the STL inserter *result*.

```
PatchCallback *cb() const;
```

Returns the PatchCallback object associated with this PatchBlock.

4.1.4 PatchEdge

Declared in: PatchCFG.h

The PatchEdge class is a wrapper of ParseAPI's Edge class (has-a), which joins two PatchBlocks in the CFG, indicating the type of control flow transfer instruction that joins the basic blocks to each other.

```
ParseAPI::Edge *edge() const;
```

Returns a ParseAPI::Edge associated with this PatchEdge.

```
PatchBlock *src();
```

Returns the source PatchBlock.

```
PatchBlock *trg();
```

Returns the target PatchBlock.

```
ParseAPI::EdgeTypeEnum type() const;
```

Returns the edge type (ParseAPI::EdgeTypeEnum, please see ParseAPI Manual).

```
bool sinkEdge() const;
```

Indicates whether this edge targets the special sink block, where a sink block is a block to which all unresolvable control flow instructions will be linked.

```
bool interproc() const;
```

Indicates whether the edge should be interpreted as interprocedural (e.g., calls, returns, direct branches under certain circumstances).

```
PatchCallback *cb() const;
```

Returns a Patchcallback object associated with this PatchEdge.

4.1.5 PatchLoop

Declared in: PatchCFG.h

The PatchLoop class is a wrapper of ParseAPI's Loop class (has-a). It represents code structure that may execute repeatedly.

```
PatchLoop* parent
```

Returns the loop which directly encloses this loop. NULL if no such loop.

```
bool containsAddress(Address addr)
```

Returns true if the given address is within the range of this loop's basic blocks.

```
bool containsAddressInclusive(Address addr)
```

Returns true if the given address is within the range of this loop's basic blocks or its children.

```
int getLoopEntries(vector<PatchBlock*>& entries);
```

Fills `entries` with the set of entry basic blocks of the loop. Return the number of the entries that this loop has

`int getBackEdges(vector<PatchEdge*> &edges)`

Sets `edges` to the set of back edges in this loop. It returns the number of back edges that are in this loop.

`bool getContainedLoops(vector<PatchLoop*> &loops)`

Returns a vector of loops that are nested under this loop.

`bool getOuterLoops(vector<PatchLoop*> &loops)`

Returns a vector of loops that are directly nested under this loop.

`bool getLoopBasicBlocks(vector<PatchBlock*> &blocks)`

Fills `blocks` with all basic blocks in the loop

`bool getLoopBasicBlocksExclusive(vector<PatchBlock*> &blocks)`

Fills `blocks` with all basic blocks in this loop, excluding the blocks of its sub loops.

`bool hasBlock(PatchBlock *b);`

Returns `true` if this loop contains basic block `b`.

`bool hasBlockExclusive(PatchBlock *b);`

Returns `true` if this loop contains basic block `b` and `b` is not in its sub loops.

`bool hasAncestor(PatchLoop *loop)`

Returns true if this loop is a descendant of the given loop.

`PatchFunction * getFunction();`

Returns the function that this loop is in.

4.1.6 PatchLoopTreeNode

Declared in: PatchCFG.h

The PatchLoopTreeNode class provides a tree interface to a collection of instances of class PatchLoop contained in a function. The structure of the tree follows the nesting relationship of the loops in a function. Each PatchLoopTreeNode contains a pointer to a loop (represented by PatchLoop), and a set of sub-loops (represented by other PatchLoopTreeNode objects). The `loop` field at the root node is always `NULL` since a function may contain multiple outer loops. The `loop` field is never `NULL` at any other node since it always corresponds to a real loop. Therefore, the outer most loops in the function are contained in the vector of `children` of the root.

Each instance of PatchLoopTreeNode is given a name that indicates its position in the hierarchy of loops. The name of each outermost loop takes the form of `loop_x`, where `x` is an integer from 1 to n, where n is the number of outer loops in the function. Each sub-loop has the name of its parent, followed by a `.y`, where `y` is 1 to m, where m is the number of sub-loops under the outer loop. For example, consider the following C function:

```
void foo() {
  int x, y, z, i;
  for (x=0; x<10; x++) {
    for (y = 0; y<10; y++)
      ...
    for (z = 0; z<10; z++)
      ...
  }
  for (i = 0; i<10; i++) {
     ...
  }
}
```

The `foo` function will have a root PatchLoopTreeNode, containing a NULL loop entry and two PatchLoopTreeNode children representing the functions outermost loops. These children would have names `loop_1` and `loop_2`, respectively representing the `x` and `i` loops. `loop_2` has no children. `loop_1` has two child PatchLoopTreeNode objects, named `loop_1.1` and `loop_1.2`, respectively representing the `y` and `z` loops.

`PatchLoop *loop;`

The PatchLoop instance it points to.

`std::vector<PatchLoopTreeNode *> children;`

The PatchLoopTreeNode instances nested within this loop.

`const char * name();`

Returns the hierarchical name of this loop.

`const char * getCalleeName(unsigned int i)`

Returns the function name of the ith callee.

`unsigned int numCallees()`

Returns the number of callees contained in this loop's body.

`bool getCallees(vector<PatchFunction *> &v);`

Fills `v` with a vector of the functions called inside this loop.

`PatchLoop * findLoop(const char *name);`

Looks up a loop by the hierarchical name

4.2 Point/Snippet Interface

4.2.1 PatchMgr

Declared in: PatchMgr.h

The PatchMgr class is the top-level class for finding instrumentation **Points**, inserting or deleting **Snippets**, and registering user-provided plugins.

```
static PatchMgrPtr create(AddrSpace* as, Instrumenter* inst = NULL,
                          PointMaker* pm = NULL);
```

This factory method creates a new PatchMgr object that performs binary code patching. It takes input three plugins, including AddrSpace *as*, Instrumenter *inst*, and PointMaker *pm*. PatchAPI uses default plugins for PointMaker and Instrumenter, if *pm* and *inst* are not specified (NULL by default).

This method returns PatchMgrPtr() if it was unable to create a new PatchMgr object.

```
Point *findPoint(Location loc, Point::Type type, bool create = true);
```

This method returns a unique Point according to a Location *loc* and a Type *type*. The Location structure is to specify a physical location of a Point (e.g., at function entry, at block entry, etc.), details of Location will be covered in Section 4.2.2. PatchAPI creates Points on demand, so if a Point is not yet created, the *create* parameter is to indicate whether to create this Point. If the Point we want to find is already created, this method simply returns a pointer to this Point from a buffer, no matter whether *create* is true or false. If the Point we want to find is not yet created, and *create* is true, then this method constructs this Point and put it in a buffer, and finally returns a Pointer to this Point. If the Point creation fails, this method also returns false. If the Point we want to find is not yet created, and *create* is false, this method returns NULL. The basic logic of finding a point can be found in the Listing 10.

Listing 10: Pseudocode of finding a point

```
if (point is in the buffer) {
  return point;
} else {
  if (create == true) {
    create point
    if (point creation fails) return NULL;
    put the point in the buffer
  } else {
    return NULL;
  }
}
```

```
template <class OutputIterator>
bool findPoint(Location loc, Point::Type type, OutputIterator outputIter,
               bool create = true);
```

This method finds a Point at a physical Location *loc* with a *type*. It adds the found Point to *outputIter* that is a STL inserter. The point is created on demand. If the Point is already created, then this method outputs a pointer to this Point from a buffer. Otherwise, the *create* parameter indicates whether to create this Point.

This method returns true if a point is found, or the *create* parameter is false; otherwise, it returns false.

```
template <class OutputIterator>
bool findPoints(Location loc, Point::Type types, OutputIterator outputIter,
                bool create = true);
```

This method finds Points at a physical Location *loc* with composite *types* that are combined using the overloaded operator "|". This function outputs Points to the STL inserter *outputIter*. The point is created on demand. If the Point is already created, then this method outputs a pointer to this Point from a buffer. Otherwise, the *create* parameter indicates whether to create this Point.

This method returns true if a point is found, or the *create* parameter is false; otherwise, it returns false.

```
template <class FilterFunc,
          class FilterArgument,
          class OutputIterator>
bool findPoints(Location loc, Point::Type types, FilterFunc filter_func,
                FilterArgument filter_arg, OutputIterator outputIter,
                bool create = true);
```

This method finds Points at a physical Location *loc* with composite *types* that are combined using the overloaded operator "|". Then, this method applies a filter functor *filter_func* with an argument *filter_ arg* on each found Point. The method outputs Points to the inserter *outputIter*. The point is created on demand. If the Point is already created, then this method returns a pointer to this Point from a buffer. Otherwise, the *create* parameter indicates whether to create this Point.

If no any Point is created, then this method returns false; otherwise, true is returned. The code below shows the prototype of an example functor.

Listing 11: Code template for the filter function in findPoint

```
template <class T>
class FilterFunc {
  public:
    bool operator()(Point::Type type, Location loc, T arg) {
      // The logic to check whether this point is what we need
      return true;
    }
};
```

In the functor FilterFunc above, programmers check each candidate Point by looking at the Point::Type, Location, and the user-specified parameter *arg*. If the return value is true, then the Point being checked will be put in the STL inserter *outputIter*; otherwise, this Point will be discarded.

```
struct Scope {
  Scope(PatchBlock *b);
  Scope(PatchFunction *f, PatchBlock *b);
  Scope(PatchFunction *f);
};
```

The Scope structure specifies the scope to find points, where a scope could be a function, or a basic block. This is quite useful if programmers don't know the exact Location, then they can use Scope as a wildcard. A basic block can be contained in multiple functions. The second constructor only specifies the block *b* in a particular function *f*.

```
template <class FilterFunc,
          class FilterArgument,
          class OutputIterator>
bool findPoints(Scope scope, Point::Type types, FilterFunc filter_func,
                FilterArgument filter_arg, OutputIterator output_iter,
                bool create = true);
```

This method finds points in a *scope* with certain *types* that are combined together by using the overloaded operator "|". Then, this method applies the filter functor *filter_func* on each found Point. It outputs Points where *filter_func* returns true to the STL inserter *output_ iter*. Points are created on demand. If some points are already created, then this method outputs pointers to them from a buffer. Otherwise, the *create* parameter indicates whether to create Points.

If no any Point is created, then this function returns false; otherwise, true is returned.

```
template <class OutputIterator>
bool findPoints(Scope scope, Point::Type types, OutputIterator output_iter, bool create = true);
```

This method finds points in a *scope* with certain *types* that are combined together by using the overloaded operator "|". It outputs the found points to the STL inserter *output_ iter*. If some points are already created, then this method outputs pointers to them from a buffer. Otherwise, the *create* parameter indicates whether to create Points.

If no any Point is created, then this method returns false; otherwise, true is returned.

```
bool removeSnippet(InstancePtr);
```

This method removes a snippet Instance.

It returns false if the point associated with this Instance cannot be found; otherwise, true is returned.

```
template <class FilterFunc,
          class FilterArgument>
bool removeSnippets(Scope scope, Point::Type types, FilterFunc filter_func,
                    FilterArgument filter_arg);
```

This method deletes ALL snippet instances at certain points in certain *scope* with certain *types*, and those points pass the test of *filter_func*.

If no any point can be found, this method returns false; otherwise, true is returned.

```
bool removeSnippets(Scope scope, Point::Type types);
```

This method deletes ALL snippet instances at certain points in certain *scope* with certain *types*.

If no any point can be found, this method returns false; otherwise, true is returned.

```
void destroy(Point *point);
```

This method is to destroy the specified *Point*.

```
AddrSpace* as() const;
PointMaker* pointMaker() const;
Instrumenter* instrumenter() const;
```

The above three functions return the corresponding plugin: AddrSpace, PointMaker, Instrumenter.

4.2.2 Point

Declared in: Point.h

The Point class is in essence a container of a list of snippet instances. Therefore, the Point class has methods similar to those in STL.

```
struct Location {
   static Location Function(PatchFunction *f);
   static Location Block(PatchBlock *b);
   static Location BlockInstance(PatchFunction *f, PatchBlock *b, bool trusted = false);
   static Location Edge(PatchEdge *e);
   static Location EdgeInstance(PatchFunction *f, PatchEdge *e);
   static Location Instruction(PatchBlock *b, Address a);
   static Location InstructionInstance(PatchFunction *f, PatchBlock *b, Address a);
   static Location InstructionInstance(PatchFunction *f, PatchBlock *b, Address a,
                                       InstructionAPI::Instruction::Ptr i,
                                       bool trusted = false);
   static Location EntrySite(PatchFunction *f, PatchBlock *b, bool trusted = false);
   static Location CallSite(PatchFunction *f, PatchBlock *b);
   static Location ExitSite(PatchFunction *f, PatchBlock *b);
};
```

The Location structure uniquely identifies the physical location of a point. A Location object plus a Point::Type value uniquely identifies a point, because multiple Points with different types can exist at the same physical location. The Location structure provides a set of static functions to create an object of Location, where each function takes the corresponding CFG structures to identify a physical location. In addition, some functions above (e.g., InstructionInstance) takes input the *trusted* parameter that is to indicate PatchAPI whether the CFG structures passed in is trusted. If the *trusted* parameter is false, then PatchAPI would have additional checking to verify the CFG structures passed by users, which causes nontrivial overhead.

```
enum Point::Type {
  PreInsn,
  PostInsn,
  BlockEntry,
  BlockExit,
  BlockDuring,
  FuncEntry,
  FuncExit,
  FuncDuring,
  EdgeDuring,
  PreCall,
  PostCall,
  OtherPoint,
  None,
  InsnTypes = PreInsn | PostInsn,
  BlockTypes = BlockEntry | BlockExit | BlockDuring,
  FuncTypes = FuncEntry | FuncExit | FuncDuring,
  EdgeTypes = EdgeDuring,
  CallTypes = PreCall | PostCall
};
```

The enum Point::Type specifies the logical point type. Multiple enum values can be OR-ed to form a composite type. For example, the composite type of “PreCall | BlockEntry | FuncExit” is to specify a set of points with the type PreCall, or BlockEntry, or FuncExit.

```
typedef std::list<InstancePtr>::iterator instance_iter;
instance_iter begin();
instance_iter end();
```

The method begin() returns an iterator pointing to the beginning of the container storing snippet Instances, while the method end() returns an iterator pointing to the end of the container (past the last element).

```
InstancePtr pushBack(SnippetPtr);
InstancePtr pushFront(SnippetPtr);
```

Multiple instances can be inserted at the same Point. We maintain the instances in an ordered list. The pushBack method is to push the specified Snippet to the end of the list, while the pushFront method is to push to the front of the list.

Both methods return the Instance that uniquely identifies the inserted snippet.

```
bool remove(InstancePtr instance);
```

This method removes the given snippet *instance* from this Point.

```
void clear();
```

This method removes all snippet instances inserted to this Point.

```
size_t size();
```

Returns the number of snippet instances inserted at this Point.

```
Address addr() const;
```

Returns the address associated with this point, if it has one; otherwise, it returns 0.

```
Type type() const;
```

Returns the Point type of this point.

`bool empty() const;`

Indicates whether the container of instances at this Point is empty or not.

`PatchFunction* getCallee();`

Returns the function that is invoked at this Point, which should have Point::Type of Point::PreCall or Point::PostCall. It there is not a function invoked at this point, it returns NULL.

`const PatchObject* obj() const;`

Returns the PatchObject where the Point resides.

`const InstructionAPI::Instruction::Ptr insn() const;`

Returns the Instruction where the Point resides.

`PatchFunction* func() const;`

Returns the function where the Point resides.

`PatchBlock* block() const;`

Returns the PatchBlock where the Point resides.

`PatchEdge* edge() const;`

Returns the Edge where the Point resides.

`PatchCallback *cb() const;`

Returns the PatchCallback object that is associated with this Point.

`static bool TestType(Point::Type types, Point::Type type);`

This static method tests whether a set of *types* contains a specific *type*.

`static void AddType(Point::Type& types, Point::Type type);`

This static method adds a specific *type* to a set of *types*.

`static void RemoveType(Point::Type& types, Point::Type trg);`

This static method removes a specific *type* from a set of *types*.

4.2.3 Instance

Declared in: Point.h

The Instance class is a representation of a particular snippet inserted at a particular point. If a Snippet is inserted to N points or to the same point for N times ($N > 1$), then there will be N Instances.

`bool destroy();`

This method destroys the snippet Instance itself.

`Point* point() const;`

Returns the Point where the Instance is inserted.

`SnippetPtr snippet() const;`

Returns the Snippet. Please note that, the same Snippet may have multiple instances inserted at different Points or the same Point.

4.3 Callback Interface

4.3.1 PatchCallback

Declared in: PatchCallback.h

The PatchAPI CFG layer may change at runtime due to program events (e.g., a program loading additional code or overwriting its own code with new code). The `PatchCallback` interface allows users to specify callbacks they wish to occur whenever the PatchAPI CFG changes.

```
virtual void destroy_cb(PatchBlock *);
virtual void destroy_cb(PatchEdge *);
virtual void destroy_cb(PatchFunction *);
virtual void destroy_cb(PatchObject *);
```

Programmers implement the above virtual methods to handle the event of destroying a PatchBlock, a PatchEdge, a PatchFunction, or a PatchObject respectively. All the above methods will be called before corresponding object destructors are called.

```
virtual void create_cb(PatchBlock *);
virtual void create_cb(PatchEdge *);
virtual void create_cb(PatchFunction *);
virtual void create_cb(PatchObject *);
```

Programmers implement the above virtual methods to handle the event of creating a PatchBlock, a PatchEdge, a PatchFunction, or a PatchObject respectively. All the above methods will be called after the objects are created.

```
virtual void split_block_cb(PatchBlock *first, PatchBlock *second);
```

Programmers implement the above virtual method to handle the event of splitting a PatchBlock as a result of a new edge being discovered. The above method will be called after the block is split.

```
virtual void remove_edge_cb(PatchBlock *, PatchEdge *, edge_type_t);
virtual void add_edge_cb(PatchBlock *, PatchEdge *, edge_type_t);
```

Programmers implement the above virtual methods to handle the events of removing or adding an PatchEdge respectively. The method remove_edge_cb will be called before the event triggers, while the method add_edge_cb will be called after the event triggers.

```
virtual void remove_block_cb(PatchFunction *, PatchBlock *);
virtual void add_block_cb(PatchFunction *, PatchBlock *);
```

Programmers implement the above virtual methods to handle the events of removing or adding a PatchBlock respectively. The method remove_block_cb will be called before the event triggers, while the method add_block_cb will be called after the event triggers.

```
virtual void create_cb(Point *pt);
virtual void destroy_cb(Point *pt);
```

Programmers implement the create_cb method above, which will be called after the Point *pt* is created. And, programmers implement the destroy_cb method, which will be called before the point *pt* is deleted.

```
virtual void change_cb(Point *pt, PatchBlock *first, PatchBlock *second);
```

Programmers implement this method, which is to be invoked after a block is split. The provided Point belonged to the first block and is being moved to the second.

5 Modification API Reference

This section describes the modification interface of PatchAPI. While PatchAPI's main goal is to allow users to insert new code into a program, a secondary goal is to allow safe modification of the original program code as well.

To modify the binary, a user interacts with the `PatchModifier` class to manipulate a PatchAPI CFG. CFG modifications are then instantiated as new code by the PatchAPI. For example, if PatchAPI is being used as part of Dyninst, executing a `finalizeInsertionSet` will generate modified code.

The three key benefits of the PatchAPI modification interface are abstraction, safety, and interactivity. We use the CFG as a mechanism for transforming binaries in a platform-independent way that requires no instruction-level knowledge by the user. These transformations are limited to ensure that the CFG can always be used to instantiate code, and thus the user can avoid unintended side-effects of modification. Finally, modifications to the CFG are represented in that CFG, allowing users to iteratively combine multiple CFG transformations to achieve their goals.

Since modification can modify the CFG, it may invalidate any analyses the user has performed over the CFG. We suggest that users take advantage of the callback interface described in Section 4.3.1 to update any such analysis information.

The PatchAPI modification capabilities are currently in beta; if you experience any problems or bugs, please contact `bugs@dyninst.org`.

Many of these methods return a boolean type; true indicates a successful operation, and false indicates a failure. For methods that return a pointer, a `NULL` return value indicates a failure.

```
bool redirect(PatchEdge *edge, PatchBlock *target);
```

Redirects the edge specified by `edge` to a new target specified by `target`. In the current implementation, the edge may not be indirect.

```
PatchBlock *split(PatchBlock *orig, Address addr,
                  bool trust = false,
                  Address newlast = (Address) -1);
```

Splits the block specified by `orig`, creating a new block starting at `addr`. If `trust` is true, we do not verify that `addr` is a valid instruction address; this may be useful to reduce overhead. If `newlast` is not -1, we use it as the last instruction address of the first block. All Points are updated to belong to the appropriate block. The second block is returned.

```
bool remove(std::vector<PatchBlock *> &blocks, bool force = true)
```

Removes the blocks specified by `blocks` from the CFG. If `force` is true, blocks are removed even if they have incoming edges; this may leave the CFG in an unsafe state but may be useful for reducing overhead.

```
bool remove(PatchFunction *func)
```

Removes `func` and all of its non-shared blocks from the CFG; any shared blocks remain.

```
class InsertedCode {
  typedef boost::shared_ptr<...> Ptr;
  PatchBlock *entry();
  const std::vector<PatchEdge *> &exits();
  const std::set<PatchBlock *> &blocks();
}

InsertedCode::Ptr insert(PatchObject *obj, SnippetPtr snip, Point *point);
InsertedCode::Ptr insert(PatchObject *obj, void *start, unsigned size);
```

Methods for inserting new code into a CFG. The `InsertedCode` structure represents a CFG subgraph generated by inserting new code; the graph has a single entry point and multiple exits, represented by edges to the sink node. The first `insert` call takes a PatchAPI Snippet structure and a Point that is used to generate that Snippet; the point is only passed through to the snippet code generator and thus may be `NULL` if the snippet does not use Point information. The second `insert` call takes a raw code buffer.

6 Plugin API Reference

This section describes the various plugin interfaces for extending PatchAPI. We expect that most users should not have to ever explicitly use an interface from this section; instead, they will use plugins previously implemented by PatchAPI developers.

As with the public interface, all objects and methods in this section are in the "Dyninst::PatchAPI" namespace.

6.1 AddrSpace

Declared in: AddrSpace.h

The AddrSpace class represents the address space of a **Mutatee**, where it contains a collection of **PatchObjects** that represent shared libraries or a binary executable. In addition, programmers implement some memory management interfaces in the AddrSpace class to determine the type of the code patching - 1st party, 3rd party, or binary rewriting.

```
virtual bool write(PatchObject* obj, Address to, Address from, size_t size);
```

This method copies *size*-byte data stored at the address *from* on the **Mutator** side to the address *to* on the **Mutatee** side. The parameter *to* is the relative offset for the PatchObject *obj*, if the instrumentation is for binary rewriting; otherwise *to* is an absolute address.

If the write operation succeeds, this method returns true; otherwise, false.

```
virtual Address malloc(PatchObject* obj, size_t size, Address near);
```

This method allocates a buffer of *size* bytes on the **Mutatee** side. The address *near* is a relative address in the object *obj*, if the instrumentation is for binary rewriting; otherwise, *near* is an absolute address, where this method tries to allocate a buffer near the address *near*.

If this method succeeds, it returns a non-zero address; otherwise, it returns 0.

```
virtual Address realloc(PatchObject* obj, Address orig, size_t size);
```

This method reallocates a buffer of *size* bytes on the **Mutatee** side. The original buffer is at the address *orig*. This method tries to reallocate the buffer near the address *orig*, where *orig* is a relative address in the PatchObject *obj* if the instrumentation is for binary rewriting; otherwise, *orig* is an absolute address.

If this method succeeds, it returns a non-zero address; otherwise, it returns 0.

`virtual bool free(PatchObject* obj, Address orig);`

This method deallocates a buffer on the **Mutatee** side at the address *orig*. If the instrumentation is for binary rewriting, then the parameter *orig* is a relative address in the object *obj*; otherwise, *orig* is an absolute address.

If this method succeeds, it returns true; otherwise, it returns false.

`virtual bool loadObject(PatchObject* obj);`

This method loads a PatchObject into the address space. If this method succeeds, it returns true; otherwise, it returns false.

`typedef std::map<const ParseAPI::CodeObject*, PatchObject*> AddrSpace::ObjMap;`

`ObjMap& objMap();`

Returns a set of mappings from ParseAPI::CodeObjects to PatchObjects, where PatchObjects in all mappings represent all binary objects (either executable or libraries loaded) in this address space.

`PatchObject* executable();`

Returns the PatchObject of the executable of the **Mutatee**.

`PatchMgrPtr mgr();`

Returns the PatchMgr's pointer, where the PatchMgr contains this address space.

6.2 Snippet

Declared in: Snippet.h

The Snippet class allows programmers to customize their own snippet representation and the corresponding mini-compiler to translate the representation into the binary code.

`static Ptr create(Snippet* a);`

Creates an object of the Snippet.

```
virtual bool generate(Point *pt, Buffer &buf);
```

Users should implement this virtual function for generating binary code for the snippet.

Returns false if code generation failed catastrophically. Point *pt* is an in-param that identifies where the snippet is being generated. Buffer *buf* is an out-param that holds the generated code.

6.3 Command

Declared in: Command.h

The Command class represents an instrumentation request (e.g., snippet insertion or removal), or an internal logical step in the code patching (e.g., install instrumentation).

```
virtual bool run() = 0;
```

Executes the normal operation of this Command.

It returns true on success; otherwise, it returns false.

```
virtual bool undo() = 0;
```

Undoes the operation of this Command.

```
virtual bool commit();
```

Implements the transactional semantics: all succeed, or all fail. Basically, it performs such logic:

```
if (run()) {
  return true;
} else {
  undo();
  return false;
}
```

6.4 BatchCommand

Declared in: Command.h

The BatchCommand class inherits from the Command class. It is actually a container of a list of Commands that will be executed in a transaction: all Commands will succeed, or all will fail.

```
typedef std::list<CommandPtr> CommandList;

CommandList to_do_;
CommandList done_;
```

> This class has two protected members *to_do_* and *done_*, where *to_do_* is a list of Commands to execute, and *done_* is a list of Commands that are executed.

```
virtual bool run();
virtual bool undo();
```

> The method run() of BatchCommand invokes the run() method of each Command in *to_do_* in order, and puts the finished Commands in *done_*. The method undo() of BatchCommand invokes the undo() method of each Command in *done _* in order.

```
void add(CommandPtr command);
```

> This method adds a Command into *to_do_*.

```
void remove(CommandList::iterator iter);
```

> This method removes a Command from *to_do_*.

6.5 Instrumenter

Declared in: Command.h

The Instrumenter class inherits BatchCommand to encapsulate the core code patching logic, which includes binary code generation. Instrumenter would contain several logical steps that are individual Commands.

```
CommandList user_commands_;
```

This class has a protected data member *user_commands_* that contains all Commands issued by users, e.g., snippet insertion. This is to facilitate the implementation of the instrumentation engine.

```
static InstrumenterPtr create(AddrSpacePtr as);
```

Returns an instance of Instrumenter, and it takes input the address space *as* that is going to be instrumented.

```
virtual bool replaceFunction(PatchFunction* oldfunc, PatchFunction* newfunc);
```

Replaces a function *oldfunc* with a new function *newfunc*.

It returns true on success; otherwise, it returns false.

```
virtual bool revertReplacedFunction(PatchFunction* oldfunc);
```

Undoes the function replacement for *oldfunc*.

It returns true on success; otherwise, it returns false.

```
typedef std::map<PatchFunction*, PatchFunction*> FuncModMap;
```

The type FuncModMap contains mappings from an PatchFunction to another PatchFunction.

```
virtual FuncModMap& funcRepMap();
```

Returns the FuncModMap that contains a set of mappings from an old function to a new function, where the old function is replaced by the new function.

```
virtual bool wrapFunction(PatchFunction* oldfunc, PatchFunction* newfunc, string name);
```

Replaces all calls to *oldfunc* with calls to wrapper *newfunc* (similar to function replacement). However, we create a copy of original using the *name* that can be used to call the original. The wrapper code would look like follows:

```
void *malloc_wrapper(int size) {
  // do stuff
  void *ret = malloc_clone(size);
  // do more stuff
  return ret;
}
```

This interface requires the user to give us a name (as represented by clone) for the original function. This matches current techniques and allows users to use indirect calls (function pointers).

```
virtual bool revertWrappedFunction(PatchFunction* oldfunc);
```

Undoes the function wrapping for *oldfunc*.

It returns true on success; otherwise, it returns false.

```
virtual FuncModMap& funcWrapMap();
```

The type FuncModMap contains mappings from the original PatchFunction to the wrapper PatchFunction.

```
bool modifyCall(PatchBlock *callBlock, PatchFunction *newCallee,
                PatchFunction *context = NULL);
```

Replaces the function that is invoked in the basic block *callBlock* with the function *newCallee*. There may be multiple functions containing the same *callBlock*, so the *context* parameter specifies in which function the *callBlock* should be modified. If *context* is NULL, then the *callBlock* would be modified in all PatchFunctions that contain it. If the *newCallee* is NULL, then the *callBlock* is removed.

It returns true on success; otherwise, it returns false.

```
bool revertModifiedCall(PatchBlock *callBlock, PatchFunction *context = NULL);
```

Undoes the function call modification for *oldfunc*. There may be multiple functions containing the same *callBlock*, so the *context* parameter specifies in which function the *callBlock* should be modified. If *context* is NULL, then the *callBlock* would be modified in all PatchFunctions that contain it.

It returns true on success; otherwise, it returns false.

```
bool removeCall(PatchBlock *callBlock, PatchFunction *context = NULL);
```

Removes the *callBlock*, where a function is invoked. There may be multiple functions containing the same *callBlock*, so the *context* parameter specifies in which function the *callBlock* should be modified. If *context* is NULL, then the *callBlock* would be modified in all PatchFunctions that contain it.

It returns true on success; otherwise, it returns false.

```
typedef map<PatchBlock*,        // B  : A call block
            map<PatchFunction*, // F_c: Function context
                PatchFunction*> // F  : The function to be replaced
           > CallModMap;
```

The type CallModMap maps from B -> F_c -> F, where B identifies a call block, and F_c identifies an (optional) function context for the replacement. If F_c is not specified, we use NULL. F specifies the replacement callee; if we want to remove the call entirely, we use NULL.

```
CallModMap& callModMap();
```

Returns the CallModMap for function call replacement / removal.

```
AddrSpacePtr as() const;
```

Returns the address space associated with this Instrumenter.

6.6 Patcher

Declared in: Command.h

The class Patcher inherits from the class BatchCommand. It accepts instrumentation requests from users, where these instrumentation requests are Commands (e.g., snippet insertion). Furthermore, Patcher implicitly adds an instance of Instrumenter to the end of the Command list to generate binary code and install the instrumentation.

```
Patcher(PatchMgrPtr mgr)
```

The constructor of Patcher takes input the relevant PatchMgr *mgr*.

```
virtual bool run();
```

Performs the same logic as BatchCommand::run(), except that this function implicitly adds an internal Command – Instrumenter, which is executed after all other Commands in the *to_ do_*.

6.7 CFGMaker

Declared in: CFGMaker.h

The CFGMaker class is a factory class that constructs the above CFG structures (PatchFunction, PatchBlock, and PatchEdge). The methods in this class are used by PatchObject. Programmers can extend PatchFunction, PatchBlock and PatchEdge by annotating their own data, and then use this class to instantiate these CFG structures.

```
virtual PatchFunction* makeFunction(ParseAPI::Function* func, PatchObject* obj);
virtual PatchFunction* copyFunction(PatchFunction* func, PatchObject* obj);

virtual PatchBlock* makeBlock(ParseAPI::Block* blk, PatchObject* obj);
virtual PatchBlock* copyBlock(PatchBlock* blk, PatchObject* obj);

virtual PatchEdge* makeEdge(ParseAPI::Edge* edge, PatchBlock* src,
                            PatchBlock* trg, PatchObject* obj);
virtual PatchEdge* copyEdge(PatchEdge* edge, PatchObject* obj);
```

Programmers implement the above virtual methods to instantiate a CFG structure (either a PatchFunction, a PatchBlock, or a PatchEdge) or to copy (e.g., when forking a new process).

6.8 PointMaker

Declared in: Point.h

The PointMaker class is a factory class that constructs instances of the Point class. The methods of the PointMaker class are invoked by PatchMgr's findPoint methods. Programmers can extend the Point class, and then implement a set of virtual methods in this class to instantiate the subclasses of Point.

```
PointMaker(PatchMgrPtr mgr);
```

The constructor takes input the relevant PatchMgr *mgr*.

```
virtual Point *mkFuncPoint(Point::Type t, PatchMgrPtr m, PatchFunction *f);
virtual Point *mkFuncSitePoint(Point::Type t, PatchMgrPtr m, PatchFunction *f, PatchBlock *b);
virtual Point *mkBlockPoint(Point::Type t, PatchMgrPtr m, PatchBlock *b, PatchFunction *context);
virtual Point *mkInsnPoint(Point::Type t, PatchMgrPtr m, PatchBlock *, Address a,
                           InstructionAPI::Instruction::Ptr i, PatchFunction *context);
virtual Point *mkEdgePoint(Point::Type t, PatchMgrPtr m, PatchEdge *e, PatchFunction *context);
```

Programmers implement the above virtual methods to instantiate the subclasses of Point.

6.9 Default Plugin

6.10 PushFrontCommand and PushBackCommand

Declared in: Command.h

The class PushFrontCommand and the class PushBackCommand inherit from the Command class. They are to insert a snippet to a point. A point maintains a list of snippet instances. PushFrontCommand would add the new snippet instance to the front of the list, while PushBackCommand would add to the end of the list.

```
static Ptr create(Point* pt, SnippetPtr snip);
```

This static method creates an object of PushFrontCommand or PushBackCommand.

```
InstancePtr instance();
```

Returns a snippet instance that is inserted at the point.

6.11 RemoveSnippetCommand

Declared in: Command.h

The class RemoveSnippetCommand inherits from the Command class. It is to delete a snippet Instance.

```
static Ptr create(InstancePtr instance);
```

This static function creates an instance of RemoveSnippetCommand.

6.12 RemoveCallCommand

Declared in: Command.h

The class RemoveCallCommand inherits from the class Command. It is to remove a function call.

```
static Ptr create(PatchMgrPtr mgr, PatchBlock* call_block, PatchFunction* context = NULL);
```

> This static method takes input the relevant PatchMgr *mgr*, the *call_ block* that contains the function call to be removed, and the PatchFunction *context*. There may be multiple PatchFunctions containing the same *call_ block*. If the *context* is NULL, then the *call_ block* would be deleted from all PatchFunctions that contains it; otherwise, the *call_ block* would be deleted only from the PatchFuncton *context*.

6.13 ReplaceCallCommand

Declared in: Command.h

The class ReplaceCallCommand inherits from the class Command. It is to replace a function call with another function.

```
static Ptr create(PatchMgrPtr mgr, PatchBlock* call_block,
                  PatchFunction* new_callee, PatchFunction* context);
```

> This Command replaces the *call_ block* with the new PatchFunction *new_ callee*. There may be multiple functions containing the same *call_ block*, so the *context* parameter specifies in which function the *call_ block* should be replaced. If *context* is NULL, then the *call_ block* would be replaced in all PatchFunctions that contains it.

6.14 ReplaceFuncCommand

Declared in: Command.h

The class ReplaceFuncCommand inherits from the class Command. It is to replace an old function with the new one.

```
static Ptr create(PatchMgrPtr mgr, PatchFunction* old_func, PatchFunction* new_func);
```

> This Command replaces the old PatchFunction *old_func* with the new PatchFunction *new_func*.

A PatchAPI for Dyninst Programmers

The PatchAPI is a Dyninst component and as such is accessible through the main Dyninst interface (BPatch objects). However, the PatchAPI instrumentation and CFG models differ from the Dyninst models in several critical ways that should be accounted for by users. This section summarizes those differences and describes how to access PatchAPI abstractions from the DyninstAPI interface.

A.1 Differences Between DyninstAPI and PatchAPI

The DyninstAPI and PatchAPI differ primarily in their CFG representations and instrumentation point abstractions. In general, PatchAPI is more powerful and can better represent complex binaries (e.g., highly optimized code or malware). In order to maintain backwards compatibility, the DyninstAPI interface has not been extended to match the PatchAPI. As a result, there are some caveats.

The PatchAPI uses the same CFG model as the ParseAPI. The primary representation is an interprocedural graph of basic blocks and edges. Functions are defined on top of this graph as collections of blocks. **A block may be contained by more than one function;** we call this the *shared block* model. Functions are defined to have a single entry block, and functions may overlap if they contain the same blocks. Call and return edges exist in the graph, and therefore traversing the graph may enter different functions. PatchAPI users may specify instrumenting a particular block within a particular function (a *block instance*) by specifying both the block and the function.

The DyninstAPI uses a historic CFG model. The primary representation is the function. Functions contain a intraprocedural graph of blocks and edges. As a result, a basic block belongs to only one function, but two blocks from different functions may be *clones* of each other. No interprocedural edges are represented in the graph, and thus traversing the CFG from a particular function is guaranteed to remain inside that function.

As a result, multiple DyninstAPI blocks may map to the same PatchAPI block. If instrumenting a particular block instance is desired, the user should provide both the DyninstAPI basic block and function.

In addition, DyninstAPI uses a *module* abstraction, where a `BPatch_module` represents a collection of functions from a particular source file (for the executable) or from an entire library (for all libraries). PatchAPI, like ParseAPI, instead uses an *object* representation, where a `PatchObject` object represents a collection of functions from a file on disk (executable or libraries).

The instrumentation point (*instPoint*) models also differ between DyninstAPI and PatchAPI. We classify an instPoint either as a *behavior* point (e.g., function entry) or *location* point (e.g., a particular instruction). PatchAPI fully supports both of these models, with the added extension that a location point explicitly specifies whether instrumentation will execute before or after the corresponding location. Dyninst does not support the behavior model, instead mapping behavior instPoints to a corresponding instruction. For example, if a user requests a function entry instPoint they instead receive an instPoint for the first instruction in the function. These may not always be the same (see Bernat_AWAT). In addition, location instPoints represent an instruction, and the user must later specify whether they wish to instrument before or after that instruction.

As a result, there are complications for using both DyninstAPI and PatchAPI. We cannot emphasize enough, though, that users *can combine DyninstAPI and PatchAPI* with some care. Doing so offers several benefits:

- The ability to extend legacy code that is written for DyninstAPI.
- The ability to use the DyninstAPI extensions and plugins for PatchAPI, including snippet-based or dynC-based code generation and our instrumentation optimizer.

We suggest the following best practices to be followed when coding for PatchAPI via Dyninst:

- For legacy code, do not attempt to map between DyninstAPI instPoints and PatchAPI instPoints. Instead, use DyninstAPI CFG objects to acquire PatchAPI CFG objects, and use a `PatchMgr` (acquired through a `BPatch_addressSpace`) to look up PatchAPI instPoints.
- For new code, acquire a `PatchMgr` directly from a `BPatch_addressSpace` and use its methods to look up both CFG objects and instPoints.

A.2 PatchAPI accessor methods in Dyninst

To access a PatchAPI class from a Dyninst class, use the `PatchAPI::convert` function, as in the following example:

```
BPatch_basicBlock *bp_block = ...;
PatchAPI::PatchBlock *block = PatchAPI::convert(bp_block);
```

We support the following mappings, where all PatchAPI objects are within the `Dyninst::PatchAPI` namespace:

From	To	Comments
`BPatch_function`	`PatchFunction`	
`BPatch_basicBlock`	`PatchBlock`	See above.
`BPatch_edge`	`PatchEdge`	See above.
`BPatch_module`	`PatchObject`	See above.
`BPatch_image`	`PatchMgr`	
`BPatch_addressSpace`	`PatchMgr`	
`BPatch_snippet`	`Snippet`	

We do not support a direct mapping between `BPatch_points` and `Points`, as the failure of Dyninst to properly represent behavior instPoints leads to confusing results. Instead, use the PatchAPI point lookup methods.