Paradyn Parallel Performance Tools

Dyninst Programmer's Guide

Release 10.1
May 2019

Department of Computer Sciences
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
github.com/dyninst/dyninst

Contents

1 Introduction

The normal cycle of developing a program is to edit the source code, compile it, and then execute the resulting binary. However, sometimes this cycle can be too restrictive. We may wish to change the program while it is executing or after it has been linked, thus avoiding the process of re-compiling, re-linking, or even re-executing the program to change the binary. At first, this may seem like a bizarre goal, however, there are several practical reasons why we may wish to have such a system. For example, if we are measuring the performance of a program and discover a performance problem, it might be necessary to insert additional instrumentation into the program to understand the problem. Another application is performance steering; for large simulations, computational scientists often find it advantageous to be able to make modifications to the code and data while the simulation is executing.

This document describes an Application Program Interface (API) to permit the insertion of code into a computer application that is either running or on disk. The API for inserting code into a running application, called dynamic instrumentation, shares much of the same structure as the API for inserting code into an executable file or library, known as static instrumentation. The API also permits changing or removing subroutine calls from the application program. Binary code changes are useful to support a variety of applications including debugging, performance monitoring, and to support composing applications out of existing packages. The goal of this API is to provide a ma-chine independent interface to permit the creation of tools and applications that use runtime and static code patching. The API and a simple test application are described in [1]. This API is based on the idea of dynamic instrumentation described in [3].
The key features of this interface are the abilities to:

- Insert and change instrumentation in a running program.
- Insert instrumentation into a binary on disk and write a new copy of that binary back to disk.
- Perform static and dynamic analysis on binaries and processes.

The goal of this API is to keep the interface small and easy to understand. At the same time, it needs to be sufficiently expressive to be useful for a variety of applications. We accomplished this goal by providing a simple set of abstractions and a way to specify which code to insert into the application . [1]

[1] To generate more complex code, extra (initially un-called) subroutines can be linked into the application program, and calls to these subroutines can be inserted at runtime via this interface.

2 Abstractions

The DyninstAPI library provides an interface for instrumenting and working with binaries and processes. The user writes a *mutator*, which uses the DyninstAPI library to operate on the application. The process that contains the *mutator* and DyninstAPI library is known as the *mutator process*. The *mutator process* operates on other processes or on-disk binaries, which are known as *mutatees*.

The API is based on abstractions of a program. For dynamic instrumentation, it can be based on the state while in execution. The two primary abstractions in the API are *points* and *snippets*. A *point* is a location in a program where instrumentation can be inserted. A *snippet* is a representation of some executable code to be inserted into a program at a point. For example, if we wished to record the number of times a procedure was invoked, the *point* would be entry point of the procedure, and the *snippets* would be a statement to increment a counter. *snippets* can include conditionals and function calls.

Mutatees are represented using an *address space* abstraction. For dynamic instrumentation, the *address space* represents a process and includes any dynamic libraries loaded with the process. For static instrumentation, the *address space* includes a disk executable and includes any dynamic library files on which the executable depends. The *address space* abstraction is extended by *process* and *binary* abstractions for dynamic and static instrumentation. The *process* abstraction represents information about a running process such as threads or stack state. The *binary* abstraction represents information about a binary found on disk.

The code and data represented by an *address space* is broken up into *function* and *variable* abstractions. *Functions* contain *points*, which specify locations to insert instrumentation. *Functions* also contain a *control flow graph* abstraction, which contains information about basic blocks, edges, loops, and instructions. If the *mutatees* contains debug information, DyninstAPI will also provide abstractions about *variable* and *function types*, *local variables*, *function parameters*, and *source code line inforamtion*. The collection of *functions* and *variable* in a *mutatees* is represented as an *image*.

The API includes a simple type system based on structural equivalence. If mutatee programs have been compiled with debugging symbols and the symbols are in a format that Dyninst understands, type checking is performed on code to be inserted into the mutatee. See Section 4.28 for a complete description of the type system.

Due to language constructs or compiler optimizations, it may be possible for multiple functions to *overlap* (that is, share part of the same function body) or for a single function to have multiple *entry points*. In practice, it is impossible to determine the difference between multiple overlapping functions and a single function with multiple entry points. The DyninstAPI uses a model where each function (BPatch_function object) has a single entry point, and multiple functions may overlap (share code). We guarantee that instrumentation inserted in a particular function is only executed in the context of that function, even if instrumentation is inserted into a location that exists in multiple functions.

3 Examples

To illustrate the ideas of the API, we present several short examples that demonstrate how the API can be used. The full details of the interface are presented in the next section. To prevent confusion, we refer to the application process or binary that is being modified as the mutatee, and the program that uses the API to modify the application as the mutator. The mutator is a separate process from the application process.

The examples in this section are simple code snippets, not complete programs. Appendix A - Complete Examples provides several examples of complete Dyninst programs.

3.1 Instrumenting A Function

A mutator program must create a single instance of the class `BPatch` . This object is used to access functions and information that are global to the library. It must not be destroyed until the mutator has completely finished using the library. For this example, we assume that the mutator program has declared a global variable called `bpatch` of class `BPatch` .
All instrumentation is done with a `BPatch_addressSpace` object, which allows us to write codes that work for both dynamic and static instrumentation. During initialization we use either `BPatch_process` to attach to or create a process, or `BPatch_binaryEdit` to open a file on disk. When instrumentation is completed, we will either run the `BPatch_process`, or write the `BPatch_binaryEdit` back onto the disk.

The mutator first needs to identify the application to be modified. If the process is already in execution, this can be done by specifying the executable file name and process id of the application as arguments in order to create an instance of a process object:

```
BPatch_process *appProc = bpatch.processAttach(name, processId);
```

If the mutator is opening a file for static binary rewriting, it executes:

```
BPatch_binaryEdit *appBin = bpatch.openBinary(pathname);
```

The above statements create either a `BPatch_process` object or `BPatch_binaryEdit` object, depending on whether Dyninst is doing dynamic or static instrumentation. The instrumentation and analysis code can be made agnostic towards static or dynamic modes by using a `BPatch_addressSpace` object. Both `BPatch_process` and `BPatch_binaryEdit` inherit from `BPatch_addressSpace`, so we can use cast operations to move between the two:

```
BPatch_process *appProc = static_cast<BPatch_process *>(appAddrSpace)
−or−
BPatch_binaryEdit *appBin = static_cast<BPatch_binaryEdit *>(appAddrSpace)
```

Similarly, all instrumentation commands can be performed on a `BPatch_addressSpace` object, allowing similar codes to be used between dynamic instrumentation and binary rewriting:

```
BPatch_addressSpace *app = appProc;
−or−
BPatch_addressSpace *app = appBin;
```

Once the address space has been created, the mutator defines the snippet of code to be inserted and identifies where the points should be inserted.

If the mutator wants to instrument the entry point of `InterestingProcedure`, it should get a `BPatch_function` from the application's `BPatch_image`, and get the entry `BPatch_point`

from that function:

```
std::vector<BPatch_function *> functions;
std::vector<BPatch_point *> *points;
BPatch_image *appImage = app->getImage();
appImage->findFunction("InterestingProcedure", functions);
points = functions[0]->findPoint(BPatch_locEntry);
```

The mutator also needs to construct the instrumentation that it will insert at the `BPatch_point`. It can do this by allocating an integer in the application to store instrumentation results, and then creating a `BPatch_snippet` to increment that integer :

```
BPatch_variableExpr *intCounter =
   app->malloc(*(appImage->findType("int")));

BPatch_arithExpr addOne(BPatch_assign, *intCounter,
   BPatch_arithExpr(BPatch_plus, *intCounter, BPatch_constExpr(1)));
```

The mutator can set the `BPatch_snippet` to be run at the `BPatch_point` by executing an `insertSnippet` call:

```
app->insertSnippet(addOne, *points);
```

Finally, the mutator should either continue the mutate process and wait for it to finish, or write the resulting binary onto the disk, depending on whether it is doing dynamic or static instrumentation:

```
appProc->continueExecution();
while (!appProc->isTerminated()) {
   bpatch.waitForStatusChange();
} -or-
appBin->writeFile(newPath);
```

A complete example can be found in Appendix A - Complete Examples.

3.2 Binary Analysis

This example will illustrate how to use Dyninst to iterate over a function's control flow graph and inspect instructions. These are steps that would usually be part of a larger data flow or control flow analysis. Specifically, this example will collect every basic block in a function, iterate over them, and count the number of instructions that access memory.

Unlike the previous instrumentation example, this example will analyze a binary file on disk. Bear in mind, these techniques can also be applied when working with processes. This example makes use of InstructionAPI, details of which can be found in the InstructionAPI Reference Manual. Similar to the above example, the mutator will start by creating a BPatch object and opening a file to operate on:

```
BPatch bpatch;
BPatch_binaryEdit *binedit = bpatch.openFile(pathname);
```

The mutator needs to get a handle to a function to do analysis on. This example will look up a function by name; alternatively, it could have iterated over every function in `BPatch_image` or `BPatch_module`:

```
BPatch_image *appImage = binedit->getImage();

std::vector<BPatch_function *> funcs;
image->findFunction(\InterestingProcedure", funcs);
```

A function's control flow graph is represented by the `BPatch_flowGraph` class. The `BPatch_flowGraph` contains, among other things, a set of `BPatch_basicBlock` objects connected by `BPatch_edge` objects. This example will simply collect a list of the basic blocks in `BPatch_flowGraph` and iterate over each one:

```
BPatch_flowGraph *fg = funcs[0]->getCFG();

std::set<BPatch_basicBlock *> blocks;
fg->getAllBasicBlocks(blocks);
```

Given an `Instruction` object, which is described in the InstructionAPI Reference Manual, we can query for properties of this instruction. InstructionAPI has numerous methods for inspecting the memory accesses, registers, and other properties of an instruction. This example simply checks whether this instruction accesses memory:

```
std::set<BPatch_basicBlock *>::iterator block_iter;
for (block_iter = blocks.begin(); block_iter != blocks.end(); ++block_
iter) {
   BPatch_basicBlock *block = *block_iter;
   std::vector<Dyninst::InstructionAPI::Instruction::Ptr> insns;
   block->getInstructions(insns);
}
```

Given an `Instruction` object, which is described in the InstructionAPI Reference Manual, we can query for properties of this instruction. InstructionAPI has numerous methods for inspecting the memory accesses, registers, and other properties of an instruction. This example simply checks whether this instruction accesses memory:

```
std::vector<Dyninst::InstructionAPI::Instruction::Ptr>::iterator insn_
iter;
for (insn_iter = insns.begin(); insn_iter != insns.end(); ++insn_iter)
{
   Dyninst::InstructionAPI::Instruction::Ptr insn = *insn_iter;
   if (insn->readsMemory() || insn->writesMemory()) {
      insns_access_memory++;
   }
}
```

3.3 Instrumenting Memory Access

There are two snippets useful for memory access instrumentation: `BPatch_effectiveAddressExpr` and `BPatch_bytesAccessedExpr`. Both have nullary con-structors; the result of the snippet depends on the instrumentation point where the snippet is insert-ed. `BPatch_effectiveAddressExpr` has type void*, while `BPatch_bytesAccessedExpr` has type int.

These snippets may be used to instrument a given instrumentation point if and only if the

point has memory access information attached to it. In this release the only way to create instrumentation points that have memory access information attached is via `BPatch_function.findPoint(const std::set<BPatch_opCode>&)`. For example, to instrument all the loads and stores in a function named InterestingProcedure with a call to printf, one may write:

```
BPatch_addressSpace *app = ...;
BPatch_image *appImage = proc->getImage();

// We're interested in loads and stores
std::set<BPatch_opCode> axs;
axs.insert(BPatch_opLoad);
axs.insert(BPatch_opStore);

// Scan the function InterestingProcedure and create instrumentation points
std::vector<BPatch_function*> funcs;
appImage->findFunction(\InterestingProcedure", funcs);
std::vector<BPatch_point*>* points = funcs[0]->findPoint(axs);

// Create the printf function call snippet
std::vector<BPatch_snippet*> printfArgs;
BPatch_snippet *fmt = new BPatch_constExpr("Access at:  printfArgs.push_-
back(fmt);
BPatch_snippet *eae = new BPatch_effectiveAddressExpr();
printfArgs.push_back(eae);

// Find the printf function
std::vector<BPatch_function *> printfFuncs;
appImage->findFunction("printf", printfFuncs);

// Construct the function call snippet
BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

// Insert the snippet at the instrumentation points
app->insertSnippet(printfCall, *points);
```

4 Interface

This section describes functions in the API. The API is organized as a collection of C++ classes. The primary classes are `BPatch` , `BPatch_process`, `BPatch_binaryEdit`, `BPatch_thread`, `BPatch_image`, `BPatch_point`, and `BPatch_snippet`. The API also uses a template class called std::vector. This class is based on the Standard Template Library (STL) vector class.

4.1 CLASS BPatch

The BPatch class represents the entire Dyninst library. There can only be one instance of this class at a time. This class is used to perform functions and obtain information that is not specific to a particular thread or image.

```
std::vector<BPatch_process*> *getProcesses()
```

Returns the list of processes that are currently defined. This list includes processes that were directly created by calling processCreate/processAttach, and indirectly by the UNIX fork or the Windows CreateProcess system call. It is up to the user to delete this vector when they are done with it.

```
BPatch_process *processAttach(const char *path, int pid, BPatch_hybrid-
Mode mode=BPatch_normalMode)
BPatch_process *processCreate(const char *path, const char *argv[], const
char **envp = NULL, int stdin_fd=0, int stdout_fd=1, int stderr_fd=2, BPatch_-
hybridMode mode=BPatch_normalMode)
```

Each of these functions returns a pointer to a new instance of the BPatch_process class. The path parameter needed by these functions should be the pathname of the executable file containing the process image. The processAttach function returns a BPatch_process associated with an existing process. On Linux platforms the path parameter can be NULL since the executable image can be derived from the process pid. Attaching to a process puts it into the stopped state. The processCreate function creates a new process and returns a new BPatch_process associated with it. The new process is put into a stopped state be-fore executing any code.

The stdin_fd, stdout_fd, and stderr_fd parameters are used to set the standard input, output, and error of the child process. The default values of these parameters leave the input, output, and error to be the same as the mutator process. To change these values, an open UNIX file descriptor (see open(1)) can be passed.

The `mode` parameter is used to select the desired level of code analysis. Activating hybrid code analysis causes Dyninst to augment its static analysis of the code with run-time code discovery techniques. There are three modes: `BPatch_normalMode`, `BPatch_-exploratoryMode`, and `BPatch_defensiveMode`. Normal mode enables the regular static analysis features of Dyninst. Exploratory mode and defensive mode enable addtional dynamic features to correctly analyze programs that contain uncommon code patterns, such as malware. Exploratory mode is primarily oriented towards analyzing dynamic control transfers, while defensive mode additionally aims to tackle code obfuscation and self-modifying code. Both of these modes are still experimental and should be used with caution. Defensive mode is only supported on Windows.

Defensive mode has been tested on normal binaries (binaries that run correctly under normal mode), as well as some simple, packed executables (self-decrypting or decompress-

ing). More advanced forms of code obfuscation, such as self-modifying code, have not been tested recently. The traditional Dyninst interface may be used for instrumentation of binaries in defensive mode, but in the case of highly obfuscated code, this interface may prove to be ineffective due to the lack of a complete view of control flow at any given point. Therefore, defensive mode also includes a set of callbacks that enables instrumentation to be performed as new code is discovered. Due to the fact that recent efforts have focused on simpler forms of obfuscation, these callbacks have not been tested in detail. The next release of Dyninst will target more advanced uses of defensive mode.

`BPatch_binaryEdit *openBinary(const char *path, bool openDependencies = false)`

This function opens the executable file or library file pointed to by path for binary rewriting. If `openDependencies` is true then Dyninst will also open all shared libraries that path depends on. Upon success, this function returns a new instance of a `BPatch_binaryEdit` class that represents the opened file and any dependent shared libraries. This function re-turns NULL in the event of an error.

`bool pollForStatusChange()`

This is useful for a mutator that needs to periodically check on the status of its managed threads and does not want to check each process individually. It returns true if there has been a change in the status of one or more threads that has not yet been reported by either `isStopped` or `isTerminated`.

`void setDebugParsing (bool state)`

Turn on or off the parsing of debugger information. By default, the debugger information (produced by the –g compiler option) is parsed on those platforms that support it. However, for some applications this information can be quite large. To disable parsing this information, call this method with a value of `false` prior to creating a process.

`bool parseDebugInfo()`

Return true if debugger information parsing is enabled, or `false` otherwise.

`void setTrampRecursive (bool state)`

Turn on or off trampoline recursion. By default, any snippets invoked while another snippet is active will not be executed. This is the safest behavior, since recursively-calling snippets can cause a program to take up all available system resources and die. For example, adding instrumentation code to the start of printf, and then calling printf from that snippet will result in infinite recursion.

This protection operates at the granularity of an instrumentation point. When snippets are first inserted at a point, this flag determines whether code will be created with recursion protection. Changing the flag is ***not*** retroactive, and inserting more snippets will not change the recursion protection of the point. Recursion protection increases the overhead of instrumentation points, so if there is no way for the snippets to call themselves, calling

this method with the parameter `true` will result in a performance gain. The default value of this flag is `false`.

`bool isTrampRecursive ()`

Return whether trampoline recursion is enabled or not. `True` means that it is enabled.

`void setTypeChecking(bool state)`

Turn on or off type-checking of snippets. By default type-checking is turned on, and an attempt to create a snippet that contains type conflicts will fail. Any snippet expressions created with type-checking off have the type of their left operand. Turning type-checking off, creating a snippet, and then turning type-checking back on is similar to the type cast operation in the C programming language.

`bool isTypeChecked()`

Return `true` if type-checking of snippets is enabled, or `false` otherwise.

`bool waitForStatusChange()`

This function waits until there is a status change to some thread that has not yet been re-ported by either isStopped or isTerminated, and then returns true. It is more efficient to call this function than to call pollForStatusChange in a loop, because waitForStatusChange blocks the mutator process while waiting.

`void setDelayedParsing (bool)`

Turn on or off delayed parsing. When it is activated Dyninst will initially parse only the symbol table information in any new modules loaded by the program, and will postpone more thorough analysis (instrumentation point analysis, variable analysis, and discovery of new functions in stripped binaries). This analysis will automatically occur when the infor-mation is necessary.

Users which require small run-time perturbation of a program should not delay parsing; the overhead for analysis may occur at unexpected times if it is triggered by internal Dyninst behavior. Users who desire instrumentation of a small number of functions will benefit from delayed parsing.

`bool delayedParsingOn()`

Return `true` if delayed parsing is enabled, or `false` otherwise.

`void setInstrStackFrames(bool)`

Turn on and off stack frames in instrumentation. When on, Dyninst will create stack frames around instrumentation. A stack frame allows Dyninst or other tools to walk

a call stack through instrumentation, but introduces overhead to instrumentation. The default is to not create stack frames.

`bool getInstrStackFrames()`

Return `true` if instrumentation will create stack frames, or `false` otherwise.

`void setMergeTramp (bool)`

Turn on or off inlined tramps. Setting this value to `true` will make each base trampoline have all of its mini-trampolines inlined within it. Using inlined mini-tramps may allow instrumentation to execute faster, but inserting and removing instrumentation may take more time. The default setting for this is `true`.

`bool isMergeTramp ()`

This returns the current status of inlined trampolines. A value of `true` indicates that trampolines are inlined.

`void setSaveFPR (bool)`

Turn on or off floating point saves. Setting this value to false means that floating point registers will never be saved, which can lead to large performance improvements. The default value is `true`. Setting this flag may cause incorrect program behavior if the instrumentation does clobber floating point registers, so it should only be used when the user is positive this will never happen.

`bool isSaveFPROn ()`

This returns the current status of the floating point saves. `True` means we are saving float-ing points based on the analysis for the given platform.

`void setBaseTrampDeletion(bool)`

If `true`, we delete the base tramp when the last corresponding minitramp is deleted. If `false`, we leave the base tramp in. The default value is `false`.

`bool baseTrampDeletion()`

Return `true` if base trampolines are set to be deleted, or `false` otherwise.

`void setLivenessAnalysis(bool)`

If `true`, we perform register liveness analysis around an instPoint before inserting instrumentation, and we only save registers that are live at that point. This can lead to faster run-time speeds, but at the expense of slower instrumentation time. The default value is

`true`.

`bool livenessAnalysisOn()`

Return `true` if liveness analysis is currently enabled.

`void getBPatchVersion(int &major, int &minor, int &subminor)`

Return Dyninst's version number. The major version number will be stored in major, the minor version number in minor, and the subminor version in subminor. For example, under Dyninst 5.1.0, this function will return 5 in major, 1 in minor, and 0 in subminor.

`int getNotificationFD()`

Returns a file descriptor that is suitable for inclusion in a call to select(). Dyninst will write data to this file descriptor when it to signal a state change in the process. `BPatch::pollForStatusChange` should then be called so that Dyninst can handle the state change. This is useful for applications where the user does not want to block in `BPatch::waitForStatusChange`. The file descriptor will reset when the user calls `BPatch::pollForStatusChange`.

`BPatch_type *createArray(const char *name, BPatch_type *ptr, unsigned int low, unsigned int hi)`

Create a new array type. The name of the type is name, and the type of each element is ptr. The index of the first element of the array is low, and the last is high. The standard rules of type compatibility, described in Section 4.28, are used with arrays created using this function.

`BPatch_type *createEnum(const char *name, std::vector<char *> &element-Names, std::vector<int> &elementIds)`
`BPatch_type *createEnum(const char *name, std::vector<char *> &element-Names)`

Create a new enumerated type. There are two variations of this function. The first one is used to create an enumerated type where the user specifies the identifier (int) for each element. In the second form, the system specifies the identifiers for each element. In both cases, a vector of character arrays is passed to supply the names of the elements of the enumerated type. In the first form of the function, the number of element in the element-Names and elementIds vectors must be the same, or the type will not be created and this function will return NULL. The standard rules of type compatibility, described in Section 4.28, are used with enums created using this function.

`BPatch_type *createScalar(const char *name, int size)`

Create a new scalar type. The name field is used to specify the name of the type, and the size parameter is used to specify the size in bytes of each instance of the type. No additional information about this type is supplied. The type is compatible with other

scalars with the same name and size.

```
BPatch_type *createStruct(const char *name, std::vector<char *> &field-
Names, std::vector<BPatch_type *> &fieldTypes)
```

Create a new structure type. The name of the structure is specified in the name parameter. The fieldNames and fieldTypes vectors specify fields of the type. These two vectors must have the same number of elements or the function will fail (and return NULL). The standard rules of type compatibility, described in Section 4.28, are used with structures created using this function. The size of the structure is the sum of the size of the elements in the fieldTypes vector.

```
BPatch_type *createTypedef(const char *name, BPatch_type *ptr)
```

Create a new type called name and having the type ptr.

```
BPatch_type *createPointer(const char *name, BPatch_type *ptr)
BPatch_type *createPointer(const char *name, BPatch_type *ptr, int size)
```

Create a new type, named name, which points to objects of type ptr. The first form creates a pointer whose size is equal to sizeof(void*)on the target platform where the mutatee is running. In the second form, the size of the pointer is the value passed in the size parameter.

```
BPatch_type *createUnion(const char *name, std::vector<char *> &field-
Names, std::vector<BPatch_type *> &fieldTypes)
```

Create a new union type. The name of the union is specified in the name parameter. The fieldNames and fieldTypes vectors specify fields of the type. These two vectors must have the same number of elements or the function will fail (and return NULL). The size of the union is the size of the largest element in the fieldTypes vector.

4.2 Callbacks

The following functions are intended as a way for API users to be informed when an error or significant event occurs. Each function allows a user to register a handler for an event. The return code for all callback registration functions is the address of the handler that was previously registered (which may be NULL if no handler was previously registered). For backwards compatibility reasons, some callbacks may pass a `BPatch_thread` object when a `BPatch_process` may be more appropriate. A `BPatch_thread` may be converted into a `BPatch_process` using `BPatch_thread::getProcess()`.

4.2.1 Asynchronous Callbacks

```
typedef void (*BPatchAsyncThreadEventCallback)( BPatch_process *proc, BPatch_-
thread *thread)
bool registerThreadEventCallback(BPatch_asyncEventType type, BPatchAsync-
ThreadEventCallback cb)
```

```
bool removeThreadEventCallback(BPatch_asyncEventType type, BPatch_Async-
ThreadEventCallback cb)
```

The type parameter can be either one of BPatch_threadCreateEvent or BPatch_thread-DestroyEvent. Different callbacks can be registered for different values of type.

4.2.2 Code Discovery Callbacks

```
typedef void (*BPatchCodeDiscoveryCallback)( BPatch_Vector<BPatch_func-
tion*> &newFuncs, BPatch_Vector<BPatch_function*> &modFuncs)
bool registerCodeDiscoveryCallback(BPatchCodeDiscoveryCallback cb)
bool removeCodeDiscoveryCallback(BPatchCodeDiscoveryCallback cb)
```

This callback is invoked whenever previously un-analyzed code is discovered through runtime analysis, and delivers a vector of functions whose analysis have been modified and a vector of functions that are newly discovered.

4.2.3 Code Overwrite Callbacks

```
typedef void (*BPatchCodeOverwriteBeginCallback)(
   BPatch_Vector<BPatch_basicBlock*> &overwriteLoopBlocks);
typedef void (*BPatchCodeOverwriteEndCallback)(
   BPatch_Vector<std::pair<Dyninst::Address,int> > &deadBlocks,
   BPatch_Vector<BPatch_function*> &owFuncs,
   BPatch_Vector<BPatch_function*> &modFuncs,
   BPatch_Vector<BPatch_function*> &newFuncs)
bool registerCodeOverwriteCallbacks(
   BPatchCodeOverwriteBeginCallback cbBegin,
   BPatchCodeOverwriteEndCallback cbEnd)
```

Register a callback at the beginning and end of overwrite events. Only invoke if Dyninst's hybrid analysis mode is set to `BPatch_defensiveMode`.

The `BPatchCodeOverwriteBeginCallback` callback allows the user to remove any in-strumentation when the program starts writing to a code page, which may be desirable as instrumentation cannot be removed during the overwrite loop's execution, and any break-point instrumentation will dramatically slow the loop's execution.

The `BPatchCodeOverwriteEndCallback` callback delivers the effects of the overwrite loop when it is done executing. In many cases no code will have changed.

4.2.4 Dynamic Calls

```
typedef void (*BPatchDynamicCallSiteCallback)(
   BPatch_point *at_point, BPatch_function *called_function);
bool registerDynamicCallCallback(BPatchDynamicCallSiteCallback cb);
bool removeDynamicCallCallback(BPatchDynamicCallSiteCallback cb);
```

The registerDynamicCallCallback interface will not automatically instrument any dynamic call site. To make sure the call back function is called, the user needs to explicitly in-strument dynamic call sites. One way to achieve this goal is to first get instrumentation points representing dynamic call sites and then call BPatch_point::monitorCalls with a NULL input parameter.

4.2.5 Dynamic Libraries

```
typedef void (*BPatchDynLibraryCallback)(BPatch_thread *thr,
   BPatch_object *obj, bool loaded);
BPatchDynLibraryCallback registerDynLibraryCallback(
   BPatchDynLibraryCallback func)
```

Note that in versions previous to 9.1, BPatchDynLibraryCallback's signature took a BPatch_module instead of a BPatch_object.

4.2.6 Errors

```
enum BPatchErrorLevel {
   BPatchFatal, BPatchSerious, BPatchWarning, BPatchInfo };
typedef void (*BPatchErrorCallback)(
   BPatchErrorLevel severity, int number, const char * const *params)
BPatchErrorCallback registerErrorCallback(BPatchErrorCallback func)
```

This function registers the error callback function with the BPatch class. The return value is the address of the previous error callback function. Dyninst users can change the error callback during program execution (e.g., one error callback before a GUI is initialized, and a different one after). The severity field indicates how important the error is (from fatal to information/status). The number is a unique number that identifies this error message. Params are the parameters that describe the detail about an error, e.g., the process id where the error occurred. The number and meaning of params depends on the error. However, for a given error number the number of parameters returned will always be the same.

4.2.7 Exec

```
typedef void (*BPatchExecCallback)(BPatch_thread *thr)
BPatchExecCallback registerExecCallback( BPatchExecCallback func) Not im-
plemented on Windows.
```

4.2.8 Exit

```
typedef enum BPatch_exitType  NoExit, ExitedNormally, ExitedViaSignal ;
typedef void (*BPatchExitCallback)(
   BPatch_thread *proc, BPatch_exitType exit_type);
```

```
BPatchExitCallback registerExitCallback( BPatchExitCallback func)
```

Register a function to be called when a process terminates. For a normal process exit, the callback will actually be called just before the process exits, but while its process state still exists. This allows final actions to be taken on the process before it actually exits. The function `BPatch_thread::isTerminated()` will return true in this context even though the process hasn't yet actually exited. In the case of an exit due to a signal, the process will have already exited.

4.2.9 Fork

```
typedef void (*BPatchForkCallback)(BPatch_thread *parent, BPatch_thread
*child);
```

This is the prototype for the pre-fork and post-fork callbacks. The parent parameter is the parent thread, and the child parameter is a BPatch_thread in the newly created pro-cess. When invoked as a pre-fork callback, the child is `NULL`.

```
BPatchForkCallback registerPreForkCallback(
   BPatchForkCallback func) not implemented on Windows
BPatchForkCallback registerPostForkCallback(
   BPatchForkCallback func) not implemented on Windows
```

Register callbacks for pre-fork (before the child is created) and post-fork (immediately after the child is created). When a pre-fork callback is executed the child parameter will be `NULL`.

4.2.10 One Time Code

```
typedef void (*BPatchOneTimeCodeCallback)(Bpatch_thread *thr,
   void *userData, void *returnValue);
BPatchOneTimeCodeCallback registerOneTimeCodeCallback(
   BPatchOneTimeCodeCallback func)
```

The thr field contains the thread that executed the `oneTimeCode` (if thread-specific) or an unspecified thread in the process (if process-wide). The `userData` field contains the value passed to the oneTimeCode call. The `returnValue` field contains the return result of the oneTimeCode snippet.

4.2.11 Signal Handler

```
typedef void (*BPatchSignalHandlerCallback)(BPatch_point *at_point,
   long signum, std::vector<Dyninst::Address> *handlers)
bool registerSignalHandlerCallback(BPatchSignalHandlerCallback cb,
   sstd::set<long> &signal_numbers)
bool registerSignalHandlerCallback(BPatchSignalHandlerCallback cb,
   BPatch_Set<long> *signal_numbers)
```

```
bool removeSignalHandlerCallback(BPatchSignalHandlerCallback cb);
```

This function registers the signal handler callback function with the BPatch class. The return value indicates success or failure. The signal_numbers set contains those signal numbers for which the callback will be invoked.

The at_point parameter indicates the point at which the signal/exception was raised, signum is the number of the signal/exception that was raised, and the handlers vector contains any registered handler(s) for the signal/exception. In Windows this corresponds to the stack of Structured Exception Handlers, while for Unix systems there will be at most one registered exception handler. This functionality is only fully implemented for the Windows platform.

4.2.12 Stopped Threads

```
typedef void (*BPatchStopThreadCallback)(BPatch_point *at_point, void *re-
turnValue)
```

This is the prototype for the callback that is associated with the stopThreadExpr snippet class (see Section 4.13). Unlike the other callbacks in this section, stopThread-Expr callbacks are registered during the creation of the stopThreadExpr snippet type. Whenever a stopThreadExpr snippet executes in a given thread, the snippet evaluates the calculation snippet that stopThreadExpr takes as a parameter, stops the thread's execution and invokes this callback. The at_point parameter is the BPatch_point at which the stopThreadExpr snippet was inserted, and returnValue contains the computation made by the calculation snippet.

4.2.13 User-triggered callbacks

```
typedef void (*BPatchUserEventCallback)(BPatch_process *proc, void *buf,
unsigned int bufsize); bool registerUserEventCallback(BPatchUserEventCallback
cb) bool removeUserEventCallback(BPatchUserEventCallback cb)
```

Register a callback that is executed when the user sends a message from the mutatee using the DYNINSTuserMessage function in the runtime library.

4.3 CLASS BPATCH_ADDRESSSPACE

The **BPatch_addressSpace** class is a superclass of the BPatch_process and BPatch_-binaryEdit classes. It contains functionality that is common between the two sub classes.

```
BPatch_image *getImage()
```

Return a handle to the executable file associated with this BPatch_process object.

```
bool getSourceLines(unsigned long addr, std::vector< BPatch_statement >
& lines)
```

This function returns the line information associated with the mutatee address, addr. The vector lines contain pairs of filenames and line numbers that are associated with addr. In many cases only one filename and line number is associated with an address, but certain compiler optimizations may lead to multiple filenames and lines at an address. This in-formation is only available if the mutatee was compiled with debug information.

This function returns true if it was able to find any line information at addr, or false otherwise.

```
bool getAddressRanges( const char * fileName, unsigned int lineNo, std::vector<
std::pair< unsigned long, unsigned long > > & ranges )
```

Given a filename and line number, fileName and lineNo, this function this function re-turns the ranges of mutatee addresses that implement the code range in the output parameter ranges. In many cases a source code line will only have one address range implementing it. However, compiler optimizations may transform this into multiple disjoint address ranges. This information is only available if the mutatee was compiled with debug information.

This function returns true if it was able to find any line information, false otherwise.

```
BPatch_variableExpr *malloc(int n,
   std::string name = std::string(""))
BPatch_variableExpr *malloc(const BPatch_type &type,
   std::string name = std::string(""))
```

These two functions allocate memory. Memory allocation is from a heap. The heap is not necessarily the same heap used by the application. The available space in the heap may be limited depending on the implementation. The first function, malloc(int n), allocates n bytes of memory from the heap. The second function, malloc(const BPatch_type& t), allocates enough memory to hold an object of the specified type. Using the second version is strongly encouraged because it provides additional information to permit better type checking of the passed code. If a name is specified, Dyninst will assign var_name to the variable; otherwise, it will assign an internal name. The returned memory is persistent and will not be released until BPatch_process::free is called or the application terminates.

```
BPatch_variableExpr *createVariable(Dyninst::Address addr,
   BPatch_type *type,
   std::string var_name = std::string(\"),
   BPatch_module *in_module = NULL)
```

This method creates a new variable at the given address addr in the module in_module. If a name is specified, Dyninst will assign var_name to the variable; otherwise, it will assign an internal name. The type parameter will become the type for the new variable.

When operating in binary rewriting mode, it is an error for the in_module parameter to be NULL; it is necessary to specify the module in which the variable will be created. Dyninst will then write the variable back out in the file specified by in_module.

```
bool free(BPatch_variableExpr &ptr)
```

Free the memory in the passed variable `ptr`. The programmer is responsible for verifying that all code that could reference this memory will not execute again (either by removing all snippets that refer to it, or by analysis of the program). Return `true` if the free succeeded.

```
bool getRegisters(std::vector<BPatch_register> &regs)
```

This function returns a vector of `BPatch_register` objects that represent registers available to snippet code.

```
BPatchSnippetHandle *insertSnippet(const BPatch_snippet &expr,
   BPatch_point &point,
   BPatch_callWhen when=[BPatch_callBefore| BPatch_callAfter],
   BPatch_snipptOerder order = BPatch_firstSnippet)
BPatchSnippetHandle *insertSnippet(const BPatch_snippet &expr,
   const std::vector<BPatch_point *> &points,
   BPatch_callWhen when=[BPatch_callBefore| BPatch_callAfter],
   BPatch_snippetOrder order = BPatch_firstSnippet)
```

Insert a snippet of code at the specified point. If a list of points is supplied, insert the code snippet at each point in the list. The optional when argument specifies when the snip-pet is to be called; a value of `BPatch_callBefore` indicates that the snippet should be inserted just before the specified point or points in the code, and a value of `BPatch_callAfter` indicates that it should be inserted just after them.

The order argument specifies where the snippet is to be inserted relative to any other snippets previously inserted at the same point. The values `BPatch_firstSnippet` and `BPatch_lastSnippet` indicate that the snippet should be inserted before or after all snip-pets, respectively.

It is illegal to use `BPatch_callAfter` with a `BPatch_entry point`. Use `BPatch_callBefore` when instrumenting entry points, which inserts instrumentation be-fore the first instruction in a subroutine. Likewise, it is illegal to use `BPatch_callBefore` with a `BPatch_exit` point. Use `BPatch_callAfter` with exit points. `BPatch_callAfter` inserts instrumentation at the last instruction in the subroutine. `insertSnippet` will return `NULL` when used with an illegal pair of points.

```
bool deleteSnippet(BPatchSnippetHandle *handle)
```

Remove the snippet associated with the passed handle. If the handle is not defined for the process, then deleteSnippet will return `false`.

```
void beginInsertionSet()
```

Normally, a call to `insertSnippet` immediately injects instrumentation into the mutatee. However, users may wish to insert a set of snippets as a single batch operation. This provides two benefits: First, Dyninst may insert instrumentation in a more efficient manner. Second, multiple snippets may be inserted at multiple points as a single operation, with either all snippets being inserted successfully or none. This batch insertion

mode is begun with a call to `beginInsertionSet`; after this call, no snippets are actually inserted until a corresponding call to finalizeInsertionSet. Dyninst accumulates all calls to `insertSnippet` during batch mode internally, and the returned `BPatchSnippetHandles` are filled in when `finalizeInsertionSet` is called.

Insertion sets are unnecessary when doing static binary instrumentation. Dyninst uses an implicit insertion set around all instrumentation to a static binary.

```
bool finalizeInsertionSet(bool atomic)
```

Inserts all snippets accumulated since a call to `beginInsertionSet`. If the atomic parameter is `true`, then a failure to insert any snippet results in all snippets being removed; effectively, the insertion is all-or-nothing. If the `atomic` parameter is false, then snippets are inserted individually. This function also fills in the BPatchSnippetHandle structures returned by the `insertSnippet` calls comprising this insertion set. It returns `true` on success and `false` if there was an error inserting any snippets.

Insertion sets are unnecessary when doing static binary instrumentation. Dyninst uses an implicit insertion set around all instrumentation to a static binary.

```
bool removeFunctionCall(BPatch_point &point)
```

Disable the mutatee function call at the specified location. The *point* specified must be a valid call point in the image of the mutatee. The purpose of this routine is to permit tools to alter the semantics of a program by eliminating procedure calls. The mechanism to achieve the removal is platform dependent, but might include branching over the call or replacing it with NOPs. This function only removes a function call; any parameters to the function will still be evaluated.

```
bool replaceFunction (BPatch_function &old, BPatch_function &new)
bool revertReplaceFunction (BPatch_function &old)
```

Replace all calls to user function old with calls to new. This is done by inserting instrumentation (specifically a BPatch_funcJumpExpr) into the beginning of function old such that a non-returning jump is made to function new. Returns `true` upon success, false otherwise.

```
bool replaceFunctionCall(BPatch_point &point, BPatch_function &newFunc)
```

Change the function call at the specified point to the function indicated by newFunc. The purpose of this routine is to permit runtime steering tools to change the behavior of pro-grams by replacing a call to one procedure by a call to another. Point must be a function call point. If the change was successful, the return value is true, otherwise false will be returned.

WARNING: Care must be used when replacing functions. In particular if the compiler has performed inter-procedural register allocation between the original caller/callee pair, the replacement may not be safe since the replaced function may clobber registers the compiler thought the callee left untouched. Also the signatures of the both the function be-ing replaced and the new function must be compatible.

```
bool wrapFunction(BPatch_function *old, BPatch_function *new, Dyninst::SymtabAPI::Symb
*sym) bool revertWrapFunction(BPatch_function *old)
```

Replaces all calls to function old with calls to function new. Unlike replaceFunction above, the old function can still be reached via the name specified by the provided symbol sym. Function wrapping allows existing code to be extended by new code. Consider the follow-ing code that implements a fast memory allocator for a particular size of memory allocation, but falls back to the original memory allocator (referenced by origMalloc) for all others.

```
void *origMalloc(unsigned long size);

void *fastMalloc(unsigned long size) {
   if (size == 1024)
      unsigned long ret = fastPool;
      fastPool += 1024;
      return ret;
   }
   else {
      return origMalloc(size);
   }
}
```

The symbol sym is provided by the user and must exist in the program; the easiest way to ensure it is created is to use an undefined function as shown above with the definition of origMalloc.

The following code wraps malloc with fastMalloc, while allowing functions to still ac-cess the original malloc function by calling origMalloc. It makes use of the new convert interface described in Section 5.

```
using namespace Dyninst;
using namespace SymtabAPI;
BPatch_function *malloc = appImage->findFunction(...);
BPatch_function *fastMalloc = appImage->findFunction(...);
Symtab *symtab = SymtabAPI::convert(fastMalloc->getModule());
std::vector<Symbol *> syms;
symtab->findSymbol(syms, "origMalloc",
   Symbol::ST_UNKNOWN, // Don't specify type
   mangledName, // Look for raw symbol name
   false, // Not regular expression
   false, // Don't check case
   true); // Include undefined symbols
app->wrapFunction(malloc, fastMalloc, syms[0]);
```

For a full, executable example, see Appendix A Complete Examples.

```
bool replaceCode(BPatch_point *point, BPatch_snippet *snippet)
```

This function has been removed; users interested in replacing code should instead use the PatchAPI code modification interface described in the PatchAPI manual. For information

on accessing PatchAPI abstractions from DyninstAPI abstractions, see Section 5.

```
BPatch_module * loadLibrary(const char *libname, bool reload=false)
```

For dynamic rewriting, this function loads a dynamically linked library into the process's address space. For static rewriting, this function adds a library as a library dependency in the rewritten file. In both cases Dyninst creates a new BPatch_module to represent this library. The libname parameter identifies the file name of the library to be loaded, in the standard way that dynamically linked libraries are specified on the operating system on which the API is running. This function returns a handle to the loaded library. The reload parameter is ignored and only remains for backwards compatibility.

```
bool isStaticExecutable()
```

This function returns `true` if the original file opened with this `BPatch_addressSpace` is a statically linked executable, or `false` otherwise.

```
processType getType()
```

This function returns a processType that reflects whether this address space is a `BPatch_process` or a `BPatch_binaryEdit`.

4.4 CLASS BPATCH_PROCESS

The **`BPatch_process`** class represents a running process, which includes one or more threads of execution and an address space.

```
bool stopExecution()
bool continueExecution()
bool terminateExecution()
```

These three functions change the running state of the process. `stopExecution` puts the process into a stopped state. Depending on the operating system, stopping one process may stop all threads associated with a process. `continueExecution` continues execution of the process. `terminateExecution` terminates execution of the process and will invoke the ex-it callback if one is registered. Each function returns true on success, or false for failure. Stopping or continuing a terminated thread will fail and these functions will return false.

```
bool isStopped()
int stopSignal()
bool isTerminated()
```

These three functions query the status of a process. `isStopped` returns true if the process is currently stopped. If the process is stopped (as indicated by `isStopped`), then `stopSigsnal` can be called to find out what signal caused the process to stop. `isTerminated` returns true if the process has exited. Any of these functions may be called multiple times, and call-ing them will not affect the state of the process.

```
BPatch_variableExpr *getInheritedVariable(BPatch_variableExpr &parentVar)
```

Retrieve a new handle to an existing variable (such as one created by `BPatch_process::malloc`) that was created in a parent process and now exists in a forked child process. When a process forks all existing `BPatch_variableExprs` are copied to the child process, but the Dyninst handles for these objects are not valid in the child `BPatch_process`. This function is invoked on the child process' `BPatch_process`, `parentVar` is a variable from the parent process, and a handle to a variable in the child process is returned. If `parentVar` was not allocated in the parent process, then `NULL` is returned.

```
BPatchSnippetHandle *getInheritedSnippet(
   BPatchSnippetHandle &parentSnippet)
```

This function is similar to getInheritedVariable, but operates on `BPatchSnippetHandles`. Given a child process that was created via fork and a `BPatchSnippetHandle`, `parentSnippet`, from the parent process, this function will return a handle to `parentSnippet` that is valid in the child process. If it is determined that `parentSnippet` is not associated with the parent process, then `NULL` is returned.

```
void detach(bool cont)
```

Detach from the process. The process must be stopped to call this function. Instrumentation and other changes to the process will remain active in the detached copy. The cont parameter is used to indicate if the process should be continued as a result of detaching.

Linux does not support detaching from a process while leaving it stopped. All processes are continued after detach on Linux.

```
int getPid()
```

Return the system id for the mutatee process. On UNIX based systems this is a PID. On Windows this is the HANDLE object for a process.

```
typedef enum BPatch_exitType { NoExit, ExitedNormally, ExitedViaSignal };
BPatch_exitType terminationStatus()
```

If the process has exited, `terminationStatus` will indicate whether the process exited normally or because of a signal. If the process has not exited, `NoExit` will be returned. On AIX, the reason why a process exited will not be available if the process was not a child of the Dyninst mutator; in this case, `ExitedNormally` will be returned in both normal and signal exit cases.

```
int getExitCode()
```

If the process exited in a normal way, `getExitCode` will return the associated exit code. Prior to Dyninst 8.2, `getExitCode` would return the argument passed to `exit` or the value returned by `main`; in Dyninst 8.2 and later, it returns the actual exit code as

provided by the debug interface and seen by the parent process. In particular, on Linux, this means that exit codes are normalized to the range 0-255.

```
int getExitSignal()
```

If the process exited because of a received signal, getExitSignal will return the associated signal number.

```
void oneTimeCode(const BPatch_snippet &expr)
```

Cause the snippet `expr` to be executed by the mutatee immediately. If the process is multi-threaded, the snippet is run on a thread chosen by Dyninst. If the user requires the snippet to be run on a particular thread, use the `BPatch_thread` version of this function instead. The process must be stopped to call this function. The behavior is synchronous;`oneTime-Code` will not return until after the snippet has been run in the application.

```
bool oneTimeCodeAsync(const BPatch_snippet &expr,
   void *userData = NULL)
```

This function sets up a snippet to be evaluated by the process at the next available opportunity. When the snippet finishes running Dyninst will callback any function registered through `BPatch::registerOneTimeCodeCallback`, with `userData` passed as a parameter. This function return `true` on success and `false` if it could not post the `oneTimeCode`.

If the process is multithreaded, the snippet is run on a thread chosen by Dyninst. If the user requires the snippet to be run on a particular thread, use the `BPatch_thread` version of this function instead. The behavior is asynchronous; `oneTimeCodeAsync` returns before the snippet is executed.

If the process is running when `oneTimeCodeAsync` is called, `expr` will be run immediately. If the process is stopped, then `expr` will be run when the process is continued.

```
void getThreads(std::vector<BPatch_thread *> &thrds)
```

Get the list of threads in the process.

```
bool isMultithreaded()
bool isMultithreadCapable()
```

The former returns true if the process contains multiple threads; the latter returns true if the process can create threads (e.g., it contains a threading library) even if it has not yet.

4.5 CLASS BPATCH_THREAD

The ***BPatch_thread*** class represents and controls a thread of execution that is running in a pro-cess.

`void getCallStack(std::vector<BPatch_frame>& stack)`

This function fills the given vector with current information about the call stack of the thread. Each stack frame is represented by a BPatch_frame
(see section 4.24 for information about this class).

`dynthread_t getTid()`

This function returns a platform-specific identifier for this thread. This is the identifier that is used by the threading library. For example, on pthread applications this function will return the thread's `pthread_t` value.

`Dyninst::LWP getLWP()`

This function returns a platform-specific identifier that the operating system uses to identify this thread. For example, on UNIX platforms this returns the LWP id. On Windows this returns a HANDLE object for the thread.

`unsigned getBPatchID()`

This function returns a Dyninst-specific identifier for this thread. These ID's apply only to running threads, the BPatch ID of an already terminated thread my be repeated in a new thread.

`BPatch_function *getInitialFunc()`

Return the function that was used by the application to start this thread. For example, on pthread applications this will return the initial function that was passed to pthread_create.

`unsigned long getStackTopAddr()`

Returns the base address for this thread's stack.

`bool isDeadOnArrival()`

This function returns `true` if this thread terminated execution before Dyninst was able to attach to it. Since Dyninst performs new thread detection asynchronously, it is possible for a thread to be created and destroyed before Dyninst can attach to it. When this happens, a new `BPatch_thread` is created, but `isDeadOnArrival` always returns `true` for this thread. It is illegal to perform any thread-level operations on a dead on arrival thread.

`BPatch_process *getProcess()`

Return the `BPatch_process` that contains this thread.

`void *oneTimeCode(const BPatch_snippet &expr, bool *err = NULL)`

Cause the snippet `expr` to be evaluated by the process immediately. This is similar to the `BPatch_process::oneTimeCode` function, except that the snippet is guaranteed to run only on this thread. The process must be stopped to call this function. The behavior is synchronous; `oneTimeCode` will not return until after the snippet has been run in the application.

`bool oneTimeCodeAsync(const BPatch_snippet &expr, void *userData = NULL, BpatchOneTimeCodeCallback cb = NULL)`

This function sets up the snippet expr to be evaluated by this thread at the next available opportunity. When the snippet finishes running, Dyninst will callback any function registered through `BPatch::registerOneTimeCodeCallback`, with userData passed as a parameter. This function returns true if `expr` was posted or false otherwise.

This is similar to the `BPatch_process::oneTimeCodeAsync` function, except that the snippet is guaranteed to run only on this thread. The process must be stopped to call this function. The behavior is asynchronous; `oneTimeCodeAsync` returns before the snippet is executed.

4.6 CLASS BPATCH_BINARYEDIT

The ***BPatch_binaryEdit*** class represents a set of executable files and library files for binary rewriting. `BPatch_binaryEdit` inherits from the `BPatch_addressSpace` class, where most functionality for binary rewriting is found.

`bool writeFile(const char *outFile)`

Rewrite a `BPatch_binaryEdit` to disk. The original file opened with this `BPatch_binaryEdit` is written to the current working directory with the name outFile. If any dependent libraries were also opened and have instrumentation or other modifications, then those libraries will be written to disk in the current working directory under their original names.

A rewritten dependency library should only be used with the original file that was opened for rewriting. For example, if the file a.out and its dependent library libfoo.so were opened for rewriting, and both had instrumentation inserted, then the rewritten libfoo.so should not be used without the rewritten a.out. To build a rewritten libfoo.so that can load into any process, libfoo.so must be the original file opened by BPatch::openBinary.

This function returns `true` if it successfully wrote a file, or `false` otherwise.

4.7 CLASS BPATCH_SOURCEOBJ

The ***BPatch_sourceObj*** class is the C++ superclass for the `BPatch_function`, `BPatch_module`, and `BPatch_image` classes. It provides a set of common methods for all three classes. In addition, it can be used to build a "generic" source navigator using the getObjParent and `getSourceObj` methods to get parents and children of a given level (i.e. the parent of a module is an image, and the children will be the functions).

```
enum BPatchErrorLevel {
   BPatchFatal, BPatchSerious, BPatchWarning, BPatchInfo };

enum BPatch_sourceType {
   BPatch_sourceUnknown,
   BPatch_sourceProgram,
   BPatch_sourceModule,
   BPatch_sourceFunction,
   BPatch_sourceOuterLoop,
   BPatch_sourceLoop,
   BPatch_sourceStatement };

BPatch_sourceType getSrcType()
```

Returns the type of the current source object.

```
void getSourceObj(std::vector<BPatch_sourceObj *> &objs)
```

Returns the child source objects of the current source object. For example, when called on a `BPatch_sourceProgram` object this will return objects of type `BPatch_sourceFunction`. When called on a `BPatch_sourceFunction` object it may return `BPatch_sourceOuterLoop` and `BPatch_sourceStatement` objects.

```
BPatch_sourceObj *getObjParent()
```

Return the parent source object of the current source object. The parent of a `BPatch_image` is `NULL`.

```
typedef enum BPatch_language {
   BPatch_c,
   BPatch_cPlusPlus,
   BPatch_fortran,
   BPatch_fortran77,
   BPatch_fortran90,
   BPatch_f90_demangled_stabstr,
   BPatch_fortran95,
   BPatch_assembly,
   BPatch_mixed,
   BPatch_hpf,
   BPatch_java,
   BPatch_unknownLanguage
} BPatch_language;

BPatch_language getLanguage()
```

Return the source language of the current `BPatch_sourceObject`. For programs that are written in more than one language, `BPatch_mixed` will be returned. If there is insufficient information to determine the language, `BPatch_unknownLanguage` will be returned.

4.8 CLASS BPATCH_FUNCTION

An object of this class represents a function in the application. A `BPatch_image` object (see description below) can be used to retrieve a `BPatch_function` object representing a given function.

```
std::string getName(); std::string getDemangledName();
std::string getMangledName();
std::string getTypedName();
void getNames(std::vector<std::string> &names);
void getDemangledNames(std::vector<std::string> &names);
void getMangledNames(std::vector<std::string> &names);
void getTypedNames(std::vector<std::string> &names);
```

Return name(s) of the function. The `getName` functions return the primary name; this is typically the first symbol we encounter while parsing the program; `getName` is an alias for `getDemangledName`. The `getNames` functions return all known names for the function, including any names specified by weak symbols.

```
bool getAddressRange(Dyninst::Address &start, Dyninst::Address &end)
```

Returns the bounds of the function; for non-contiguous functions, this is the lowest and highest address of code that the function includes.

```
std::vector<BPatch_localVar *> *getParams()
```

Return a vector of `BPatch_localVar` snippets that refer to the parameters of this function. The position in the vector corresponds to the position in the parameter list (starting from zero). The returned local variables can be used to check the types of functions, and can be used in snippet expressions.

```
BPatch_type *getReturnType()
```

Return the type of the return value for this function.

```
BPatch_variableExpr *getFunctionRef()
```

For platforms with complex function pointers (e.g., 64-bit PPC) this constructs and returns the appropriate descriptor.

```
std::vector<BPatch_localVar *> *getVars()
```

Returns a vector of `BPatch_localVar` objects that contain the local variables in this function. These `BPatch_localVar` s can be used as parts of snippets in instrumentation. This function requires debug information to be present in the mutatee. If Dyninst was unable to find any local variables, this function will return an empty vector. It is up to the user to free the vector returned by this function.

```
bool isInstrumentable()
```

Return `true` if the function can be instrumented, and `false` if it cannot. Various conditions can cause a function to be uninstrumentable. For example, there exists a platform-specific minimum function size beyond which a function cannot be instrumented.

```
bool isSharedLib()
```

This function returns `true` if the function is defined in a shared library.

```
BPatch_module *getModule()
```

Return the module that contains this function. Depending on whether the program was compiled for debugging or the symbol table stripped, this information may not be available. This function returns `NULL` if module information was not found.

```
char *getModuleName(char *name, int maxLen)
```

Copies the name of the module that contains this function into the buffer pointed to by `name`. Copies at most `maxLen` characters and returns a pointer to name.

```
enum BPatch_procedureLocation {
   BPatch_entry,
   BPatch_exit,
   BPatch_subroutine,
   BPatch_locInstruction,
   BPatch_locBasicBlockEntry,
   BPatch_locLoopEntry,
   BPatch_locLoopExit,
   BPatch_locLoopStartIter,
   BPatch_locLoopStartExit,
   BPatch_allLocations }

const std::vector<BPatch_point *> *findPoint(
   const BPatch_procedureLocation loc)
```

Return the `BPatch_point` or list of `BPatch_point` s associated with the procedure. It is used to select which type of points associated with the procedure will be returned. `BPatch_entry` and `BPatch_exit` request respectively the entry and exit points of the sub-routine. `BPatch_subroutine` returns the list of points where the procedure calls other procedures. If the lookup fails to locate any points of the requested type, `NULL` is returned.

```
enum BPatch_opCode { BPatch_opLoad, BPatch_opStore, BPatch_opPrefetch }
std::vector<BPatch_point *> *findPoint(const std::set<BPatch_opCode>& ops)
std::vector<BPatch_point *> *findPoint(const BPatch_Set<BPatch_opCode>&
ops)
```

Return the vector of BPatch_point s corresponding to the set of machine instruction types described by the argument. This version is used primarily for memory access instrumentation. The BPatch_opCode is an enumeration of instruction types that may be requested: BPatch_opLoad, BPatch_opStore, and BPatch_opPrefetch. Any combination of these may be requested by passing an appropriate argument set containing the desired types. The instrumentation points created by this function have additional memory access information attached to them. This allows such points to be used for memory access specific snippets (e.g. effective address). The memory access information attached is described under Memory Access classes in section 4.27.1.

```
BPatch_localVar *findLocalVar(const char *name)
```

Search the function's local variable collection for name. This returns a pointer to the local variable if a match is found. This function returns NULL if it fails to find any variables.

```
std::vector<BPatch_variableExpr *> *findVariable(const char * name)
bool findVariable(const char *name, std::vector<BPatch_variableExpr> &vars)
```

Return a set of variables matching name at the scope of this function. If no variables match in the local scope, then the global scope will be searched for matches. This function returns NULL if it fails to find any variables.

```
BPatch_localVar *findLocalParam(const char *name)
```

Search the function's parameters for a given name. A BPatch_localVar * pointer is returned if a match is found, and NULL is returned otherwise.

```
void *getBaseAddr()
```

Return the starting address of the function in the mutatee's address space.

```
BPatch_flowGraph *getCFG()
```

Return the control flow graph for the function, or NULL if this information is not available. The BPatch_flowGraph is described in section 4.16

```
bool findOverlapping(std::vector<BPatch_function *> &funcs)
```

Determine which functions overlap with the current function (see Section 2). Return true if other functions overlap the current function; the overlapping functions are added to the funcs vector. Return false if no other functions overlap the current function.

```
bool addMods(std::set<StackMod *> mods)
   implemented on x86 and x86-64
```

Apply stack modifications in mods to the current function; the StackMod class is described in section 4.25. Perform error checking, handle stack alignment requirements, and generate any modifications required for cleanup at function exit. addMods atomically adds all modifications in mods; if any mod is found to be unsafe, none of the modifications in mods will be applied.

addMods can only be used in binary rewriting mode.

Returns false if the stack modifications are unsafe or if Dyninst is unable to perform the analysis required to guarantee safety.

4.9 CLASS BPATCH_POINT

An object of this class represents a location in an application's code at which the library can insert instrumentation. A BPatch_image object (see section 4.10) is used to retrieve a BPatch_point representing a desired point in the application.

```
enum BPatch_procedureLocation {
   BPatch_entry, BPatch_exit, BPatch_subroutine,BPatch_address}
BPatch_procedureLocation getPointType()
```

Return the type of the point.

```
BPatch_function *getCalledFunction()
```

Return a BPatch_function representing the function that is called at the point. If the point is not a function call site or the target of the call cannot be determined, then this function returns NULL.

```
std::string getCalledFunctionName()
```

Returns the name of the function called at this point. This method is similar to getCalledFunction()->getName(), except in cases where DyninstAPI is running in binary rewriting mode and the called function resides in a library or object file that DyninstAPI has not opened. In these cases, Dyninst is able to determine the name of the called function, but is unable to construct a BPatch_function object.

```
BPatch_function *getFunction()
```

Returns a BPatch_function representing the function in which this point is contained.

```
BPatch_basicBlockLoop *getLoop()
```

Returns the containing BPatchbasicBlockLoop if this point is part of loop instrumentation. Returns NULL otherwise.

```
void *getAddress()
```

Return the address of the first instruction at this point.

```
bool usesTrap_NP()
```

Return `true` if inserting instrumentation at this point requires using a trap. On the x86 architecture, because instructions are of variable size, the instruction at a point may be too small for Dyninst to replace it with the normal code sequence used to call instrumentation. Also, when instrumentation is placed at points other than subroutine entry, exit, or call points, traps may be used to ensure the instrumentation fits. In this case, Dyninst replaces the instruction with a single-byte instruction that generates a trap. A trap handler then calls the appropriate instrumentation code. Since this technique is used only on some platforms, on other platforms this function always returns `false`.

```
const BPatch_memoryAccess* getMemoryAccess()
```

Returns the memory access object associated with this point. Memory access points are described in section 4.27.1

```
const std::vector<BPatchSnippetHandle *> getCurrentSnippets()
const std::vector<BPatchSnippetHandle *>
   getCurrentSnippets(BPatch_callWhen when)
```

Return the `BPatchSnippetHandle` s for the `BPatch_snippet` s that are associated with the point. If argument when is `BPatch_callBefore`, then `BPatchSnippetHandle` s for snippets installed immediately before this point will be returned. Alternatively, if when is `BPatch_callAfter`, then `BPatchSnippetHandle` s for snippets installed immediately after this point will be returned.

```
bool getLiveRegisters(std::vector<BPatch_register> &regs)
```

Fill `regs` with the registers that are live before this point (e.g., `BPatch_callBefore`). Currently returns only general purpose registers (GPRs).

```
bool isDynamic()
```

This call returns true if this is a dynamic call site (e.g. a call site where the function call is made via a function pointer).

```
void* monitorCalls(BPatch_function* func)
```

For a dynamic call site, this call instruments the call site represented by this instrumentation point with a function call. If input parameter func is not `NULL`, `func` is called at the call site as the instrumentation. If `func` is NULL, the callback function registered with `BPatch::registerDynamicCallCallback` is used for instrumentation. Under both cases, this call returns a pointer to the called function. If the instrumentation point does not represent a dynamic call site, this call returns `NULL`.

`bool stopMonitoring()`

This call returns `true` if this instrumentation point is a dynamic call site and its instrumentation is successfully removed. Otherwise, it returns `false`.

`Dyninst::InstructionAPI::Instruction::Ptr getInstructionAtPoint()`

On implemented platforms, this function returns a shared pointer to an InstructionAPI Instruction object representing the first machine instruction at this point's address. On unimplemented platforms, returns a `NULL` shared pointer.

4.10 CLASS BPATCH_IMAGE

This class defines a program image (the executable associated with a process). The only way to get a handle to a `BPatch_image` is via the `BPatch_process` member function `getImage`.

`const BPatch_point *createInstPointAtAddr (caddr_t address)`

This function has been removed because it is not safe to use. Instead, use findPoints:

`bool findPoints(Dyninst::Address addr, std::vector<BPatch_point *> &points);`

Returns a vector of `BPatch_point` s that correspond with the provided address, one per function that includes an instruction at that address. There will be one element if there is not overlapping code.

`std::vector<BPatch_variableExpr *> *getGlobalVariables()`

Return a vector of global variables that are defined in this image.

`BPatch_process *getProcess()`

Returns the `BPatch_process` associated with this image.

`char *getProgramFileName(char *name, unsigned int len)`

Fills provided buffer `name` with the program's file name up to `len` characters. The filename may include path information.

`bool getSourceObj(std::vector<BPatch_sourceObj *> &sources)`

Fill `sources` with the source objects (see section 4.7) that belong to this image. If there are no source objects, the function returns `false`. Otherwise, it returns `true`.

`std::vector<BPatch_function *> *getProcedures( bool incUninstrumentable`

```
= false)
```

Return a vector of the functions in the image. If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

```
void getObjects(std::vector<BPatch_object *> &objs)
```

Fill in a vector of objects in the image.

```
std::vector<BPatch_module *> *getModules()
```

Return a vector of the modules in the image.

```
bool getVariables(std::vector<BPatch_variableExpr *> &vars)
```

Fills `vars` with the global variables defined in this image. If there are no variable, the function returns `false`. Otherwise, it returns `true`.

```
std::vector<BPatch_function*> *findFunction(
const char *name,
std::vector<BPatch_function*> &funcs,
bool showError = true,
bool regex_case_sensitive = true,
bool incUninstrumentable = false)
```

Return a vector of `BPatch_function` s corresponding to name, or `NULL` if the function does not exist. If `name` contains a POSIX-extended regular expression, and `dont_use_regex` is false, a regular expression search will be performed on function names and matching `BPatch_function` s returned. If `showError` is true, then Dyninst will report an error via the `BPatch::registerErrorCallback` if no function is found.

If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

[NOTE: If `name` is not found to match any demangled function names in the module, the search is repeated as if `name` is a mangled function name. If this second search succeeds, functions with mangled names matching name are returned instead.]

```
std::vector<BPatch_function*> *findFunction(
std::vector<BPatch_function*> &funcs,
BPatchFunctionNameSieve bpsieve,
void *sieve_data = NULL,
int showError = 0,
bool incUninstrumentable = false)
```

Return a vector of `BPatch_function` s according to the generalized user-specified filter function `bpsieve`. This permits users to easily build sets of functions according to their own specific criteria. Internally, for each `BPatch_function` f in the image, this method

makes a call to `bpsieve(f.getName(), sieve_data)`. The user-specified function `bpsieve` is responsible for taking the `name` argument and determining if it belongs in the output vector, possibly by using extra user-provided information stored in `sieve_data`. If the name argument matches the desired criteria, `bpsieve` should return `true`. If it does not, `bpsieve` should return false.

The function bpsieve should be defined in accordance with the typedef:

```
bool (*BPatchFunctionNameSieve) (const char *name, void* sieve_data);
```

If the `incUninstrumentable` flag is set, the returned table of procedures will include uninstrumentable functions. The default behavior is to omit these functions.

`bool findFunction(Dyninst::Address addr, std::vector<BPatch_function *> &funcs)`

Find all functions that have code at the given address, `addr`. Dyninst supports functions that share code, so this method may return more than one `BPatch_function`. These functions are returned via the `funcs` output parameter. This function returns `true` if it finds any functions, `false` otherwise.

```
BPatch_variableExpr *findVariable(const char *name,
   bool showError = true)
BPatch_variableExpr *findVariable(BPatch_point &scope,
   const char *name) second form of this method is not implemented on Windows.
```

Performs a lookup and returns a handle to the named variable. The first form of the function looks up only variables of global scope, and the second form uses the passed `BPatch_point` as the scope of the variable. The returned `BPatch_variableExpr` can be used to create references (uses) of the variable in subsequent snippets. The scoping rules used will be those of the source language. If the image was not compiled with debugging symbols, this function will fail even if the variable is defined in the passed scope.

`BPatch_type *findType(const char *name)`

Performs a lookup and returns a handle to the named type. The handle can be used as an argument to `BPatch_addressSpace::malloc` to create new variables of the corresponding type.

```
BPatch_module *findModule(const char *name,
   bool substring_match = false)
```

Returns a module named `name` if present in the image. If the match fails, `NULL` is returned. If `substring_match` is true, the first module that has name as a substring of its name is returned (e.g. to find `libpthread.so.1`, search for `libpthread` with `substring_match` set to true).

`bool getSourceLines(unsigned long addr,`

```
   std::vector<BPatch_statement> & lines)
```

Given an address `addr`, this function returns a vector of pairs of filenames and line numbers at that address. This function is an alias for `BPatch_process::getSourceLines` (see section 4.4).

```
bool getAddressRanges( const char * fileName, unsigned int lineNo, std::vector<
   std::pair< unsigned long, unsigned long > > & ranges )
```

Given a file name and line number, `fileName` and `lineNo`, this function returns a list of address ranges that this source line was compiled into. This function is an alias for `BPatch_process::getAddressRanges` (see section **??**).

```
bool parseNewFunctions(std::vector<BPatch_module*> &newModules, const
   std::vector<Dyninst::Address> &funcEntryAddrs)
```

This function takes as input a list of function entry points indicated by the `funcEntryAddrs` vector, which are used to seed parsing in whatever modules they are found. All affected modules are placed in the `newModules` vector, which includes any existing modules in which new functions are found, as well as modules corresponding to new regions of the binary, for which new `BPatch_module` s are created. The return value is `true` in the event that at least one previously unknown function was identified, or `false` otherwise.

4.11 CLASS BPATCH_OBJECT

An object of this class represents the original executable or a library. It serves as a container of `BPatch_module` objects.

```
std::string name()
std::string pathName()
```

Return the name of this file; either just the file name or the fully path-qualified name.

```
Dyninst::Address fileOffsetToAddr(Dyninst::Offset offset)
```

Convert the provided offset into the file into a full address in memory.

```
struct Region
   typedef enum { UNKNOWN, CODE, DATA } type_t;
   Dyninst::Address base;
   unsigned long size;
   type_t type; ;
void regions(std::vector<Region> &regions)
```

Returns information about the address ranges occupied by this object in memory.

```
void modules(std::vector<BPatch_module *> &modules)
```

Returns the modules contained in this object.

```
std::vector<BPatch_function*> *findFunction(
   const char *name,
   std::vector<BPatch_function*> &funcs,
   bool showError = true,
   bool regex_case_sensitive = true,
   bool incUninstrumentable = false)
```

Return a vector of `BPatch_function` s corresponding to `name`, or `NULL` if the function does not exist. If name contains a POSIX-extended regular expression, and `dont_use_regex` is `false`, a regular expression search will be performed on function names and matching `BPatch_function` s returned. If `showError` is true, then Dyninst will report an error via the `BPatch::registerErrorCallback` if no function is found.

If the `incUninstrumentable` flag is set, the returned table of procedures will include un-instrumentable functions. The default behavior is to omit these functions.

[NOTE: If `name` is not found to match any demangled function names in the module, the search is repeated as if `name` is a mangled function name. If this second search succeeds, functions with mangled names matching name are returned instead.]

```
bool findPoints(Dyninst::Address addr,
   std::vector<BPatch_point *> &points);
```

Return a vector of `BPatch_point` s that correspond with the provided address, one per function that includes an instruction at that address. There will be one element if there is not overlapping code.

```
std::vector<BPatch_function*> *findFunction(
   const char *name,
   std::vector<BPatch_function*> &funcs,
   bool notify_on_failure = true,
   bool regex_case_sensitive = true,
   bool incUninstrumentable = false)
```

Return a vector of `BPatch_function` s matching `name`, or `NULL` if the function does not exist. If name contains a POSIX-extended regular expression, a regex search will be per-formed on function names, and matching `BPatch_function` s returned. [NOTE: The std::vector argument funcs must be declared fully by the user before calling this function. Passing in an uninitialized reference will result in undefined behavior.]

If the `incUninstrumentable` flag is set, the returned table of procedures will include un-instrumentable functions. The default behavior is to omit these functions.

[NOTE: If `name` is not found to match any demangled function names in the `BPatch_object`, the search is repeated as if `name` is a mangled function name. If this second search succeeds, functions with mangled names matching name are returned instead.]

4.12 CLASS BPATCH_MODULE

An object of this class represents a program module, which is part of a program's executable image. A `BPatch_module` represents a source file in either an executable or a shared library. Dyninst automatically creates a module called DEFAULT˙MODULE in each executable to hold any objects that it cannot match to a source file. `BPatch_module` objects are obtained by calling the `BPatch_image` member function `getModules`.

```
std::vector<BPatch_function*> *findFunction(
   const char *name,
   std::vector<BPatch_function*> &funcs,
   bool notify_on_failure = true,
   bool regex_case_sensitive = true,
   bool incUninstrumentable = false)
```

Return a vector of `BPatch_function` s matching `name`, or `NULL` if the function does not exist. If name contains a POSIX-extended regular expression, a regex search will be performed on function names, and matching `BPatch_function` s returned. [NOTE: The std::vector argument `funcs` must be declared fully by the user before calling this function. Passing in an uninitialized reference will result in undefined behavior.]

If the `incUninstrumentable` flag is set, the returned table of procedures will include un-instrumentable functions. The default behavior is to omit these functions.

[NOTE: If `name` is not found to match any demangled function names in the module, the search is repeated as if `name` is a mangled function name. If this second search succeeds, functions with mangled names matching name are returned instead.]

```
BPatch_Vector<BPatch_function *> *findFunctionByAddress(
   void *addr,
   BPatch_Vector<BPatch_function *> &funcs,
   bool notify_on_failure = true,
   bool incUninstrumentable = false)
```

Return a vector of `BPatch_function` s that contains `addr`, or `NULL` if the function does not exist. [NOTE: The std::vector argument `funcs` must be declared fully by the user before calling this function. Passing in an uninitialized reference will result in undefined be-havior.]

If the `incUninstrumentable` flag is set, the returned table of procedures will include un-instrumentable functions. The default behavior is to omit these functions.

```
BPatch_function *findFunctionByEntry(Dyninst::Address addr)
   Returns the function that begins at the specified address addr.
   BPatch_function *findFunctionByMangled(
   const char *mangled_name,
   bool incUninstrumentable = false)
```

Return a `BPatch_function` for the mangled function name defined in the module corresponding to the invoking `BPatch_module`, or `NULL` if it does not define the function.

If the `incUninstrumentable` flag is set, the functions searched will include un-instrumentable functions. The default behavior is to omit these functions.

```
bool getAddressRanges( char * fileName, unsigned int lineNo,
   std::vector< std::pair< unsigned long, unsigned long > > & ranges )
```

Given a filename and line number, `fileName` and `lineNo`, this function returns the ranges of mutatee addresses that implement the code range in the output parameter ranges. In many cases a source code line will only have one address range implementing it. However, compiler optimizations may turn this into multiple, disjoint address ranges. This information is only available if the mutatee was compiled with debug information.

This function may be more efficient than the `BPatch_process` version of this function. Calling `BPatch_process::getAddressRange` will cause Dyninst to parse line information for all modules in a process. If `BPatch_module::getAddressRange` is called then only the debug information in this module will be parsed.

This function returns `true` if it was able to find any line information, `false` otherwise.

```
size_t getAddressWidth()
```

Return the size (in bytes) of a pointer in this module. On 32-bit systems this function will return 4, and on 64-bit systems this function will return 8.

```
void *getBaseAddr()
```

Return the base address of the module. This address is defined as the start of the first function in the module.

```
std::vector<BPatch_function *>
   *getProcedures( bool incUninstrumentable = false )
```

Return a vector containing the functions in the module.

```
char *getFullName(char *buffer, int length)
```

Fills `buffer` with the full path name of a module, up to length characters when this information is available.

```
BPatch_hybridMode getHybridMode()
```

Return the mutator's analysis mode for the mutatee; the default mode is the normal mode.

```
char *getName(char *buffer, int len)
```

This function copies the filename of the module into `buffer`, up to `len` characters. It returns the value of the `buffer` parameter.

`unsigned long getSize()`

Return the size of the module. The size is defined as the end of the last function minus the start of the first function.

`bool getSourceLines( unsigned long addr, std::vector<BPatch_statement> & lines )`

This function returns the line information associated with the mutatee address `addr`. The vector `lines` contain pairs of filenames and line numbers that are associated with `addr`. In many cases only one filename and line number is associated with an address, but certain compiler optimizations may lead to multiple filenames and lines at an address. This information is only available if the mutatee was compiled with debug information.

This function may be more efficient than the `BPatch_process` version of this function. Calling `BPatch_process`::getSourceLines will cause Dyninst to parse line information for all modules in a process. If `BPatch_module::getSourceLines` is called then only the debug information in this module will be parsed.

This function returns `true` if it was able to find any line information at `addr`, or `false` otherwise.

`char *getUniqueString(char *buffer, int length)`

Performs a lookup and returns a unique string for this image. Returns a string the can be compared (via `strcmp`) to indicate if two images refer to the same underlying object file (i.e., executable or library). The contents of the string are implementation specific and defined to have no semantic meaning.

`bool getVariables(std::vector<BPatch_variableExpr *> &vars)`

Fill the vector `vars` with the global variables that are specified in this module. Returns `false` if no results are found and `true` otherwise.

`BpatchSnippetHandle* insertInitCallback(Bpatch_snippet& callback)`

This function inserts the snippet `callback` at the entry point of this module's `init` function (creating a new init function/section if necessary).

`BpatchSnippetHandle* insertFiniCallback(Bpatch_snippet& callback)`

This function inserts the snippet `callback` at the exit point of this module's fini function (creating a new fini function/section if necessary).

`bool isExploratoryModeOn()`

This function returns `true` if the mutator's analysis mode sets to the defensive mode or the exploratory mode.

`bool isMutatee()`

This function returns `true` if the module is the mutatee.

`bool isSharedLib()`

This function returns `true` if the module is part of a shared library.

4.13 CLASS BPATCH_SNIPPET

4.14 CLASS BPATCH_TYPE

4.15 CLASS BPATCH_VARIABLEEXPR

4.16 CLASS BPATCH_FLOWGRAPH

4.17 CLASS BPATCH_BASICBLOCK

4.18 CLASS BPATCH_EDGE

4.19 CLASS BPATCH_BASICBLOCKLOOP

4.20 CLASS BPATCH_LOOPTREENODE

4.21 CLASS BPATCH_REGISTER

4.22 CLASS BPATCH_SOURCEBLOCK

4.23 CLASS BPATCH_CBLOCK

4.24 CLASS BPATCH_FRAME

4.25 CLASS STACKMOD

4.26 CONTAINER CLASSES

4.26.1 Class std::vecotr

4.26.2 Class BPatch_Set

4.27 MEMORY ACCESS CLASSES

4.27.1 Class BPatch_memoryAccess

4.27.2 Class BPatch_addrSpec_NP

4.27.3 Class BPatch_countSpec_NP

4.28 TYPE SYSTEM

5 Using Dyninst API with the component libraries

In this section, we describe how to access the underlying component library abstractions from corresponding Dyninst abstractions. The component libraries (SymtabAPI, InstructionAPI, ParseAPI, and PatchAPI) often provide greater functionality and cleaner interfaces than Dyninst, and thus users may wish to use a mix of abstractions. In general, users may access component library abstractions via a convert function, which is overloaded and namespaced to give consistent behavior. The definitions of all component library abstractions are located in the appropriate documentation.

```
PatchAPI::PatchMgrPtr PatchAPI::convert(BPatch_addressSpace *);

PatchAPI::PatchObject *PatchAPI::convert(BPatch_object *);
ParseAPI::CodeObject *ParseAPI::convert(BPatch_object *);
SymtabAPI::Symtab *SymtabAPI::convert(BPatch_object *);

SymtabAPI::Module *SymtabAPI::convert(BPatch_module *);

PatchAPI::PatchFunction *PatchAPI::convert(BPatch_function *);
ParseAPI::Function *ParseAPI::convert(BPatch_function *);

PatchAPI::PatchBlock *PatchAPI::convert(BPatch_basicBlock *);
ParseAPI::Block *ParseAPI::convert(BPatch_basicBlock *);

PatchAPI::PatchEdge *PatchAPI::convert(BPatch_edge *);
ParseAPI::Edge *ParseAPI::convert(BPatch_edge *);

PatchAPI::Point *PatchAPI::convert(BPatch_point *, BPatch_callWhen);
PatchAPI::SnippetPtr PatchAPI::convert(BPatch_snippet *);

SymtabAPI::Type *SymtabAPI::convert(BPatch_type *);
```

6 Using the API

6.1 OVERVIEW OF MAJOR STEPS

6.2 CREATING A MUTATOR PROGRAM

6.3 SETTING UP THE APPLICATION PROGRAM(MUTATEE)

6.4 RUNNING THE MUTATOR

6.5 OPTIMIZING DYNINST PERFORMANCE

6.5.1 Optimizing Mutator Performance

6.5.2 Optimizing Mutatees Performance

A Complete Examples

In this section we show two complete examples: the programs from Section 3 and a complete Dyninst program, retee.

A.1 INSTURMENTING A FUNCTION

```
#include <stdio.h>

#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_point.h"
#include "BPatch_function.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

// Different ways to perform instrumentation
typedef enum {
    create,
    attach,
    open
} accessType_t;

// Attach, create, or open a file for rewriting
BPatch_addressSpace* startInstrumenting(accessType_t accessType,
        const char* name,
        int pid,
        const char* argv[]) {
    BPatch_addressSpace* handle = NULL;

    switch(accessType) {
        case create:
            handle = bpatch.processCreate(name, argv);
```

```
            if (!handle) { fprintf(stderr, "processCreate failed\n"); }
            break;
        case attach:
            handle = bpatch.processAttach(name, pid);
            if (!handle) { fprintf(stderr, "processAttach failed\n"); }
            break;
        case open:
            // Open the binary file and all dependencies
            handle = bpatch.openBinary(name, true);
            if (!handle) { fprintf(stderr, "openBinary failed\n"); }
            break;
    }

    return handle;
}
// Find a point at which to insert instrumentation
std::vector<BPatch_point*>* findPoint(BPatch_addressSpace* app,
                const char* name,
                BPatch_procedureLocation loc) {
    std::vector<BPatch_function*> functions;
    std::vector<BPatch_point*>* points;

    // Scan for functions named "name"
    BPatch_image* appImage = app->getImage();
    appImage->findFunction(name, functions);
    if (functions.size() == 0) {
        fprintf(stderr, "No function %s\n", name);
        return points;
    } else if (functions.size() > 1) {
        fprintf(stderr, "More than one %s; using the first one\n", name);
    }

    // Locate the relevant points
    points = functions[0]->findPoint(loc);
    return points;
}

// Create and insert an increment snippet
bool createAndInsertSnippet(BPatch_addressSpace* app,
        std::vector<BPatch_point*>* points) {
    BPatch_image* appImage = app->getImage();

    // Create an increment snippet
    BPatch_variableExpr* intCounter =
        app->malloc(*(appImage->findType("int")), "myCounter");
    BPatch_arithExpr addOne(BPatch_assign,
                            *intCounter,
                            BPatch_arithExpr(BPatch_plus,
                                *intCounter,
                                BPatch_constExpr(1)));

    // Insert the snippet
    if (!app->insertSnippet(addOne, *points)) {
        fprintf(stderr, "insertSnippet failed\n");
```

```
        return false;
    }
    return true;
}

// Create and insert a printf snippet
bool createAndInsertSnippet2(BPatch_addressSpace* app,
        std::vector<BPatch_point*>* points) {
    BPatch_image* appImage = app->getImage();

    // Create the printf function call snippet
    std::vector<BPatch_snippet*> printfArgs;
    BPatch_snippet* fmt =
        new BPatch_constExpr("InterestingProcedure called %d times\n");
    printfArgs.push_back(fmt);

    BPatch_variableExpr* var = appImage->findVariable("myCounter");
if (!var) {
        fprintf(stderr, "Could not find 'myCounter' variable\n");
        return false;
    } else {
        printfArgs.push_back(var);
    }

    // Find the printf function
    std::vector<BPatch_function*> printfFuncs;
    appImage->findFunction("printf", printfFuncs);
    if (printfFuncs.size() == 0) {
        fprintf(stderr, "Could not find printf\n");
        return false;
    }

    // Construct a function call snippet
    BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

    //  Insert the snippet
    if (!app->insertSnippet(printfCall, *points)) {
        fprintf(stderr, "insertSnippet failed\n");
        return false;
    }
    return true;
}

void finishInstrumenting(BPatch_addressSpace* app, const char* newName)
{
    BPatch_process* appProc = dynamic_cast<BPatch_process*>(app);
    BPatch_binaryEdit* appBin = dynamic_cast<BPatch_binaryEdit*>(app);

    if (appProc) {
        if (!appProc->continueExecution()) {
            fprintf(stderr, "continueExecution failed\n");
        }
        while (!appProc->isTerminated()) {
            bpatch.waitForStatusChange();
```

```
        }
    } else if (appBin) {
        if (!appBin->writeFile(newName)) {
            fprintf(stderr, "writeFile failed\n");
        }
    }
}

int main() {
    // Set up information about the program to be instrumented
    const char* progName = "InterestingProgram";
    int progPID = 42;
    const char* progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;

    // Create/attach/open a binary
    BPatch_addressSpace* app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }

    // Find the entry point for function InterestingProcedure
    const char* interestingFuncName = "InterestingProcedure";
    std::vector<BPatch_point*>* entryPoint =
        findPoint(app, interestingFuncName, BPatch_entry);
    if (!entryPoint || entryPoint->size() == 0) {
        fprintf(stderr, "No entry points for %s\n", interestingFuncName);
        exit(1);
    }

    // Create and insert instrumentation snippet
    if (!createAndInsertSnippet(app, entryPoint)) {
        fprintf(stderr, "createAndInsertSnippet failed\n");
        exit(1);
    }

    // Find the exit point of main
    std::vector<BPatch_point*>* exitPoint =
        findPoint(app, "main", BPatch_exit);
    if (!exitPoint || exitPoint->size() == 0) {
        fprintf(stderr, "No exit points for main\n");
        exit(1);
    }

    // Create and insert instrumentation snippet 2
    if (!createAndInsertSnippet2(app, exitPoint)) {
        fprintf(stderr, "createAndInsertSnippet2 failed\n");
        exit(1);
    }
    // Finish instrumentation
    const char* progName2 = "InterestingProgram-rewritten";
    finishInstrumenting(app, progName2);
```

```
}
```

A.2 BINARY ANALYSIS

```
#include <stdio.h>

#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_function.h"
#include "BPatch_flowGraph.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

// Different ways to perform instrumentation
typedef enum {
    create,
    attach,
    open
} accessType_t;

BPatch_addressSpace* startInstrumenting(accessType_t accessType,
        const char* name,
        int pid,
        const char* argv[]) {
    BPatch_addressSpace* handle = NULL;

    switch(accessType) {
        case create:
            handle = bpatch.processCreate(name, argv);
            if (!handle) { fprintf(stderr, "processCreate failed\n"); }
            break;
        case attach:
            handle = bpatch.processAttach(name, pid);
            if (!handle) { fprintf(stderr, "processAttach failed\n"); }
            break;
        case open:
            // Open the binary file and all dependencies
            handle = bpatch.openBinary(name, true);
            if (!handle) { fprintf(stderr, "openBinary failed\n"); }
            break;
    }

    return handle;
}

int binaryAnalysis(BPatch_addressSpace* app) {
    BPatch_image* appImage = app->getImage();

    int insns_access_memory = 0;
```

```
    std::vector<BPatch_function*> functions;
    appImage->findFunction("InterestingProcedure", functions);

    if (functions.size() == 0) {
        fprintf(stderr, "No function InterestingProcedure\n");
        return insns_access_memory;
    } else if (functions.size() > 1) {
        fprintf(stderr, "More than one InterestingProcedure; using the first one\n");
    }

    BPatch_flowGraph* fg = functions[0]->getCFG();

    std::set<BPatch_basicBlock*> blocks;
    fg->getAllBasicBlocks(blocks);

    for (auto block_iter = blocks.begin();
            block_iter != blocks.end();
            ++block_iter) {
        BPatch_basicBlock* block = *block_iter;
        std::vector<InstructionAPI::Instruction::Ptr> insns;
        block->getInstructions(insns);

        for (auto insn_iter = insns.begin();
                insn_iter != insns.end();
                ++insn_iter) {
            InstructionAPI::Instruction::Ptr insn = *insn_iter;
            if (insn->readsMemory() || insn->writesMemory()) {
                insns_access_memory++;
            }
        }
    }

    return insns_access_memory;
}
```

```
int main() {
    // Set up information about the program to be instrumented
    const char* progName = "InterestingProgram";
    int progPID = 42;
    const char* progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;

    // Create/attach/open a binary
    BPatch_addressSpace* app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
```

```
    }

    int memAccesses = binaryAnalysis(app);

    fprintf(stderr, "Found %d memory accesses\n", memAccesses);
}
```

A.3 INSTRUMENTING MEMORY ACCESS

```
#include <stdio.h>

#include "BPatch.h"
#include "BPatch_addressSpace.h"
#include "BPatch_process.h"
#include "BPatch_binaryEdit.h"
#include "BPatch_point.h"
#include "BPatch_function.h"

using namespace std;
using namespace Dyninst;

// Create an instance of class BPatch
BPatch bpatch;

// Different ways to perform instrumentation
typedef enum {
    create,
    attach,
    open
} accessType_t;

// Attach, create, or open a file for rewriting
BPatch_addressSpace* startInstrumenting(accessType_t accessType,
        const char* name,
        int pid,
        const char* argv[]) {
    BPatch_addressSpace* handle = NULL;

    switch(accessType) {
        case create:
            handle = bpatch.processCreate(name, argv);
            if (!handle) { fprintf(stderr, "processCreate failed\n"); }
            break;
        case attach:
            handle = bpatch.processAttach(name, pid);
            if (!handle) { fprintf(stderr, "processAttach failed\n"); }
            break;

case open:
            // Open the binary file; do not open dependencies
            handle = bpatch.openBinary(name, false);
            if (!handle) { fprintf(stderr, "openBinary failed\n"); }
            break;
    }
```

```
    return handle;
}

bool instrumentMemoryAccesses(BPatch_addressSpace* app) {
    BPatch_image* appImage = app->getImage();

    // We're interested in loads and stores
    BPatch_Set<BPatch_opCode> axs;
    axs.insert(BPatch_opLoad);
    axs.insert(BPatch_opStore);

    // Scan the function InterestingProcedure
    // and create instrumentation points
    std::vector<BPatch_function*> functions;
    appImage->findFunction("InterestingProcedure", functions);
    std::vector<BPatch_point*>* points =
        functions[0]->findPoint(axs);
    if (!points) {
        fprintf(stderr, "No load/store points found\n");
        return false;
    }

    // Create the printf function call snippet
    std::vector<BPatch_snippet*> printfArgs;
    BPatch_snippet* fmt = new BPatch_constExpr("Access at: 0x%lx\n");
    printfArgs.push_back(fmt);
    BPatch_snippet* eae = new BPatch_effectiveAddressExpr();
    printfArgs.push_back(eae);

    // Find the printf function
    std::vector<BPatch_function*> printfFuncs;
    appImage->findFunction("printf", printfFuncs);
    if (printfFuncs.size() == 0) {
        fprintf(stderr, "Could not find printf\n");
        return false;
    }

    // Construct a function call snippet
    BPatch_funcCallExpr printfCall(*(printfFuncs[0]), printfArgs);

    // Insert the snippet at the instrumentation points
    if (!app->insertSnippet(printfCall, *points)) {
        fprintf(stderr, "insertSnippet failed\n");
        return false;
    }
    return true;
}

void finishInstrumenting(BPatch_addressSpace* app, const char* newName) {
    BPatch_process* appProc = dynamic_cast<BPatch_process*>(app);
    BPatch_binaryEdit* appBin = dynamic_cast<BPatch_binaryEdit*>(app);

    if (appProc) {
        if (!appProc->continueExecution()) {
```

```
            fprintf(stderr, "continueExecution failed\n");
        }
        while (!appProc->isTerminated()) {
            bpatch.waitForStatusChange();
        }
    } else if (appBin) {
        if (!appBin->writeFile(newName)) {
            fprintf(stderr, "writeFile failed\n");
        }
    }
}

int main() {
    // Set up information about the program to be instrumented
    const char* progName = "InterestingProgram";
    int progPID = 42;
    const char* progArgv[] = {"InterestingProgram", "-h", NULL};
    accessType_t mode = create;

    // Create/attach/open a binary
    BPatch_addressSpace* app =
        startInstrumenting(mode, progName, progPID, progArgv);
    if (!app) {
        fprintf(stderr, "startInstrumenting failed\n");
        exit(1);
    }

    // Instrument memory accesses
    if (!instrumentMemoryAccesses(app)) {
        fprintf(stderr, "instrumentMemoryAccesses failed\n");
        exit(1);
    }

    // Finish instrumentation
    const char* progName2 = "InterestingProgram-rewritten";
    finishInstrumenting(app, progName2);
}
```

A.4 RETEE

The final example is a program called "re-tee." It takes three arguments: the pathname of an exe-cutable program, the process id of a running instance of the same program, and a file name. It adds code to the running program that copies to the named file all output that the program writes to its standard output file descriptor. In this way it works like "tee," which passes output along to its own standard out while also saving it in a file. The motivation for the example program is that you run a program, and it starts to print copious lines of output to your screen, and you wish to save that output in a file without having to re-run the program.

```
#include <stdio.h>
#include <fcntl.h>
#include <vector>
#include "BPatch.h"
#include "BPatch_point.h"
#include "BPatch_process.h"
#include "BPatch_function.h"
#include "BPatch_thread.h"
```

```
/*
 * retee.C
 *
 * This program (mutator) provides an example of several facets of
 * Dyninst's behavior, and is a good basis for many Dyninst
 * mutators. We want to intercept all output from a target application
 * (the mutatee), duplicating output to a file as well as the
 * original destination (e.g., stdout).
 *
 * This mutator operates in several phases. In brief:
 * 1) Attach to the running process and get a handle (BPatch_process
 *    object)
 * 2) Get a handle for the parsed image of the mutatee for function
 *    lookup (BPatch_image object)
 * 3) Open a file for output
 *    3a) Look up the "open" function
 *    3b) Build a code snippet to call open with the file name.
 *    3c) Run that code snippet via a oneTimeCode, saving the returned
 *        file descriptor
 * 4) Write the returned file descriptor into a memory variable for
 *    mutatee-side use
 * 5) Build a snippet that copies output to the file
 *    5a) Locate the "write" library call
 *    5b) Access its parameters
 *    5c) Build a snippet calling write(fd, parameters)
 *    5d) Insert the snippet at write
 * 6) Add a hook to exit to ensure that we close the file (using
 *    a callback at exit and another oneTimeCode)
 */

void usage() {
    fprintf(stderr, "Usage: retee <process pid> <filename>\n");
    fprintf(stderr, "   note: <filename> is relative to the application pro-cess.\n")
}

// We need to use a callback, and so the things that callback requires
// are made global - this includes the file descriptor snippet (see below)
BPatch_variableExpr *fdVar = NULL;

// Before we add instrumentation, we need to open the file for
// writing. We can do this with a oneTimeCode - a piece of code run at
// a particular time, rather than at a particular location.

int openFileForWrite(BPatch_process *app, BPatch_image *appImage, char *fileName) {
    // The code to be generated is:
    // fd = open(argv[2], O_WRONLY|O_CREAT, 0666);

    // (1) Find the open function
    std::vector<BPatch_function *>openFuncs;
    appImage->findFunction("open", openFuncs);
    if (openFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for open()\n");
        return -1;
```

```
    }

    // (2) Allocate a vector of snippets for the parameters to open
    std::vector<BPatch_snippet *> openArgs;

    // (3) Create a string constant expression from argv[3]
    BPatch_constExpr fileNameExpr(fileName);

    // (4) Create two more constant expressions _WRONLY|O_CREAT and 0666
    BPatch_constExpr fileFlagsExpr(O_WRONLY|O_CREAT);
    BPatch_constExpr fileModeExpr(0666);

    // (5) Push 3 & 4 onto the list from step 2, push first to last parameter.
    openArgs.push_back(&fileNameExpr);
    openArgs.push_back(&fileFlagsExpr);
    openArgs.push_back(&fileModeExpr);

    // (6) create a procedure call using function found at 1 and
    // parameters from step 5.
    BPatch_funcCallExpr openCall(*openFuncs[0], openArgs);

    // (7) The oneTimeCode returns whatever the return result from
    // the BPatch_snippet is. In this case, the return result of
    // open -> the file descriptor.
    void *openFD = app->oneTimeCode( openCall );

    // oneTimeCode returns a void *, and we want an int file handle
    return (int) (long) openFD;
}

// We have used a oneTimeCode to open the file descriptor. However,
// this returns the file descriptor to the mutator - the mutatee has
// no idea what the descriptor is. We need to allocate a variable in
// the mutatee to hold this value for future use and copy the
// (mutator-side) value into the mutatee variable.

// Note: there are alternatives to this technique. We could have
// allocated the variable before the oneTimeCode and augmented the
// snippet to do the assignment. We could also write the file
// descriptor as a constant into any inserted instrumentation.

BPatch_variableExpr *writeFileDescIntoMutatee(BPatch_process *app,
                                              BPatch_image *appImage,
                                              int fileDescriptor) {
    // (1) Allocate a variable in the mutatee of size (and type) int
    BPatch_variableExpr *fdVar = app->malloc(*appImage->findType("int"));
    if (fdVar == NULL) return NULL;

    // (2) Write the value into the variable
    // Like memcpy, writeValue takes a pointer
    // The third parameter is for functionality called "saveTheWorld",
    // which we don't worry about here (and so is false)
    bool ret = fdVar->writeValue((void *) &fileDescriptor, sizeof(int),
                                 false);
```

```
    if (ret == false) return NULL;

    return fdVar;
}

// We now have an open file descriptor in the mutatee. We want to
// instrument write to intercept and copy the output. That happens
// here.

bool interceptAndCloneWrite(BPatch_process *app,
                            BPatch_image *appImage,
                            BPatch_variableExpr *fdVar) {
    // (1) Locate the write call
    std::vector<BPatch_function *> writeFuncs;

    appImage->findFunction("write",
                           writeFuncs);
    if(writeFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for write()\n");
        return false;
    }

    // (2) Build the call to (our) write. Arguments are:
    //   ours: fdVar (file descriptor)
    //   parameter: buffer
    //   parameter: buffer size

    // Declare a vector to hold these.
    std::vector<BPatch_snippet *> writeArgs;
    // Push on the file descriptor
    writeArgs.push_back(fdVar);
    // Well, we need the buffer... but that's a parameter to the
    // function we're implementing. That's not a problem - we can grab
    // it out with a BPatch_paramExpr.
    BPatch_paramExpr buffer(1); // Second (0, 1, 2) argument
    BPatch_paramExpr bufferSize(2);
    writeArgs.push_back(&buffer);
    writeArgs.push_back(&bufferSize);

    // And build the write call
    BPatch_funcCallExpr writeCall(*writeFuncs[0], writeArgs);

    // (3) Identify the BPatch_point for the entry of write. We're
    // instrumenting the function with itself; normally the findPoint
    // call would operate off a different function than the snippet.

    std::vector<BPatch_point *> *points;
    points = writeFuncs[0]->findPoint(BPatch_entry);
    if ((*points).size() == 0) {
        return false;
    }

    // (4) Insert the snippet at the start of write
```

```
    return app->insertSnippet(writeCall, *points);

    // Note: we have just instrumented write() with a call to
    // write(). This would ordinarily be a _bad thing_, as there is
    // nothing to stop infinite recursion - write -> instrumentation
    // -> write -> instrumentation....
    // However, Dyninst uses a feature called a "tramp guard" to
    // prevent this, and it's on by default.
}

// This function is called as an exit callback (that is, called
// immediately before the process exits when we can still affect it)
// and thus must match the exit callback signature:
//
// typedef void (*BPatchExitCallback) (BPatch_thread *, BPatch_exitType)
//
// Note that the callback gives us a thread, and we want a process - but
// each thread has an up pointer.

void closeFile(BPatch_thread *thread, BPatch_exitType) {
    fprintf(stderr, "Exit callback called for process...\n");

    // (1) Get the BPatch_process and BPatch_images
    BPatch_process *app = thread->getProcess();
    BPatch_image *appImage = app->getImage();

    // The code to be generated is:
    // close(fd);

    // (2) Find close
    std::vector<BPatch_function *> closeFuncs;
    appImage->findFunction("close", closeFuncs);
    if (closeFuncs.size() == 0) {
        fprintf(stderr, "ERROR: Unable to find function for close()\n");
        return;
    }

    // (3) Allocate a vector of snippets for the parameters to open
    std::vector<BPatch_snippet *> closeArgs;

    // (4) Add the fd snippet - fdVar is global since we can't
    // get it via the callback
    closeArgs.push_back(fdVar);

    // (5) create a procedure call using function found at 1 and
    // parameters from step 3.
    BPatch_funcCallExpr closeCall(*closeFuncs[0], closeArgs);

    // (6) Use a oneTimeCode to close the file
    app->oneTimeCode( closeCall );

    // (7) Tell the app to continue to finish it off.
    app->continueExecution();
```

```
    return;
}

BPatch bpatch;

// In main we perform the following operations.
// 1) Attach to the process and get BPatch_process and BPatch_image
//    handles
// 2) Open a file descriptor
// 3) Instrument write
// 4) Continue the process and wait for it to terminate

int main(int argc, char *argv[]) {
    int pid;
    if (argc != 3) {
        usage();
        exit(1);
    }
    pid = atoi(argv[1]);

    // Attach to the program - we can attach with just a pid; the
    // program name is no longer necessary
    fprintf(stderr, "Attaching to process %d...\n", pid);
    BPatch_process *app = bpatch.processAttach(NULL, pid);

    if (!app) return -1;

    // Read the program's image and get an associated image object
    BPatch_image *appImage = app->getImage();
    std::vector<BPatch_function*> writeFuncs;

    fprintf(stderr, "Opening file %s for write...\n", argv[2]);
    int fileDescriptor = openFileForWrite(app, appImage, argv[2]);

    if (fileDescriptor == -1) {
        fprintf(stderr, "ERROR: opening file %s for write failed\n",
                argv[2]);
        exit(1);
    }

    fprintf(stderr, "Writing returned file descriptor %d into"
                    "mutatee...\n", fileDescriptor);

    // This was defined globally as the exit callback needs it.
    fdVar = writeFileDescIntoMutatee(app, appImage, fileDescriptor);
    if (fdVar == NULL) {
        fprintf(stderr, "ERROR: failed to write mutatee-side variable\n");
        exit(1);
    }

    fprintf(stderr, "Instrumenting write...\n");
    bool ret = interceptAndCloneWrite(app, appImage, fdVar);
    if (!ret) {
        fprintf(stderr, "ERROR: failed to instrument mutatee\n");
```

```
        exit(1);
    }

    fprintf(stderr, "Adding exit callback...\n");
    bpatch.registerExitCallback(closeFile);

    // Continue the execution...
    fprintf(stderr, "Continuing execution and waiting for termination\n");
    app->continueExecution();

    while (!app->isTerminated())
        bpatch.waitForStatusChange();

    printf("Done.\n");

    return 0;
}
```

B Running the Test Cases

C Common pitfalls

This appendix is designed to point out some common pitfalls that users have reported when using the Dyninst system. Many of these are either due to limitations in the current implementations, or reflect design decisions that may not produce the expected behavior from the system.

Attach followed by detach
If a mutator attaches to a mutatee, and immediately exits, the current behavior is that the mutatee is left suspended. To make sure the application continues, call detach with the appro-priate flags.

Attaching to a program that has already been modified by Dyninst
If a mutator attaches to a program that has already been modified by a previous mutator, a warning message will be issued. We are working to fix this problem, but the correct seman-tics are still being specified. Currently, a message is printed to indicate that this has been at-tempted, and the attach will fail.

Dyninst is event-driven
Dyninst must sometimes handle events that take place in the mutatee, for instance when a new shared library is loaded, or when the mutatee executes a fork or exec. Dyninst handles events when it checks the status of the mutatee, so to allow this the mutator should periodi-cally call one of the functions `BPatch::pollForStatusChange`, `BPatch::waitForStatusChange`, `BPatch_thread::isStopped`, or `BPatch_thread::isTerminated`.

Missing or out-of-date DbgHelp DLL (Windows) Dyninst requires an up-to-date DbgHelp library on Windows. See the section on Win-dows-specific architectural issues for details.

Portland Compiler Group – missing debug symbols
The Portland Group compiler (pgcc) on Linux produces debug symbols that are not read correctly by Dyninst. The binaries produced by the compiler do not contain the source file information necessary for Dyninst to assign the debug symbols to the correct module.

D References

[1] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, p. 317–329, Nov. 2000.