Paradyn Parallel Performance Tools

ParseAPI Programmer's Guide

Beta Release
Oct 2010

Computer Science Department
University of Wisconsin–Madison
Madison, WI 53711

Computer Science Department
University of Maryland
College Park, MD 20742

Email bugs@dyninst.org
Web www.dyninst.org

Contents

1 Introduction **2**

2 Abstractions **2**

3 A simple example **3**

4 The Parsing API **4**

4.1 Class CodeObject 4

4.2 Class CodeSource 6

4.3 Class CodeRegion 7

4.4 Class Function 8

4.5 Class FuncExtent 11

4.6 Class Block 11

4.7 Class Edge 13

4.8 Class EdgePredicate 14

4.9 Class ParseCallback 14

4.10 Containers 16

A Extending ParseAPI **17**

A.1 Instruction and Code Sources 17

A.2 CFG Object Factories 19

B Defensive Mode Parsing **20**

1 Introduction

A binary code parser converts the machine code representation of a program, library, or code snippet to abstractions such as the instructions, basic blocks, and functions that the binary code represents. The ParseAPI is a multi-platform library for creating such abstractions from binary code sources. The current incarnation uses the Dyninst SymtabAPI as the default binary code source; all platforms and architectures handled by the SymtabAPI are supported. The ParseAPI is designed to be easily extensible to other binary code sources. Support for parsing binary code in memory dumps or other formats requires only implementation of a small interface as described in this document.

This API provides the user with a control flow-oriented view of a binary code source. Each code object such as a program binary or library is represented as a top-level collection containing the functions, basic blocks, and edges that represent the control flow graph. A simple query interface is provided for retrieving lower level objects like functions and basic blocks through address or other attribute lookups. These objects can be used to navigate the program structure as described below.

The ParseAPI is currently released as a public beta. The interfaces described in this manual are subject to change in future versions. Feedback and comments are welcome; please email bugs@dyninst.org.

2 Abstractions

The basic representation of code in this API is the control flow graph (CFG). Binary code objects are represented as regions of contiguous bytes that, when parsed, form the nodes and edges of this graph. The following abstractions make up this CFG-oriented representation of binary code:

- BLOCK: Nodes in the CFG represent *basic blocks*: straight line sequences of instructions $I_i \ldots I_j$ where for each $i < k \leq j$, I_k postdominates I_{k-1}. Importantly, on some instruction set architectures basic blocks can *overlap* on the same address range—variable length instruction sets allow for multiple interpretations of the bytes making up the basic block.
- EDGE: Typed edges beween the nodes in the CFG represent execution control flow, such as conditional and unconditional branches, fallthrough edges, and calls and returns. The graph therefore represents both *inter-* and *intraprocedural* control flow: travseral of nodes and edges can cross the boundaries of the higher level abstractions like *functions*.
- FUNCTION: The *function* is the primary semantic grouping of code in the binary, mirroring the familiar abstraction of procedural languages like C. Functions represent the set of all basic blocks reachable from a *function entry point* through intraprocedural control flow only (that is, no calls or returns). Function entry points are determined in a variety of ways, such as hints from debugging symbols and recursive traversal along call edges.
- CODE OBJECT: A collection of distinct code regions are represented as a single code object, such as an executable or library. Code objects can normally be thought of as a single, discontiguous unique address space. However, the ParseAPI supports code objects in which the different regions have overlapping address spaces, such as UNIX archive files containing unlinked code.

- INSTRUCTION SOURCE: An instruction source describes a backing store containing binary code. A binary file, a library, a memory dump or a process's executing memory image can all be described as an instruction source, allowing parsing of a variety of binary code objects.
- CODE SOURCE: The code source implements the instruction source interface, exporting methods that can access the underlying bytes of the binary code for parsing. It also exports a number of additional helper methods that do things such as returning the location of structured exception handling routines and function symbols. Code sources are tailored to particular binary types; the ParseAPI provides a SymtabAPI-based code source that understands ELF, COFF and PE file formats.

3 A simple example

The following complete example uses the ParseAPI to parse a binary and dump its control flow graph in the Graphviz file format.

```
hash_map<Address, bool> seen;
vector<Function *> funcs;
SymtabCodeSource *sts;
CodeObject *co;

// Create a new binary code object from the filename argument
sts = new SymtabCodeSource( argv[1] );
co = new CodeObject( sts );

// Parse the binary
co->parse();

printf("digraph G {\n");

// Print the control flow graph
CodeObject::funclist & all = co->funcs();
CodeObject::funclist::iterator fit = all.begin();
for( ; fit != all.end(); ++fit) {
    Function * f = *fit;

    if(f->retstatus() == NORETURN)
        printf("\t\"%lx\" [shape=box,color=red]\n",f->addr());
    else
        printf("\t\"%lx\" [shape=box]\n",f->addr());

    Function::blocklist::iterator bit = f->blocks().begin();
    for( ; bit != f->blocks().end(); ++bit) {
        Block * b = *bit;

        // Don't revisit blocks in shared code
        if(seen.find(b->start()) != seen.end())
            continue;
```

```
            seen[b->start()] = true;

            Block::edgelist::iterator it = b->targets().begin();
            for( ; it != b->targets().end(); ++it) {
               char * s = "";
               if((*it)->type() == CALL)
                s = " [color=blue]";
               else if((*it)->type() == RET)
                s = " [color=green]";
               printf("\t\t\"%lx\" -> \"%lx\"%s\n",(*it)->src()->start(),
                (*it)->trg()->start(),s);
            }
         }
    }

    printf("}\n");

    delete co;
    delete sts;
```

4 The Parsing API

4.1 Class CodeObject

The CodeObject class describes an individual binary code object, such as an executable or library. It is the top-level container for parsing the object as well as accessing that parse data. The following API routines and data types are provided to support parsing and retrieving parsing products.

```
typedef ContainerWrapper<vector<Function*>,Function*,Function*> funclist
```

> Container for access to functions. Refer to Section 4.10 for details. Library users *must not* rely on the underlying container type of ContainerWrapper lists, as it is subject to change.

```
CodeObject(CodeSource * cs,
    CFGFactory * fact = NULL,
    ParseCallback * cb = NULL,
    bool defensiveMode = false)
```

> Constructs a new CodeObject from the provided CodeSource and optional object factory and callback handlers. Any parsing hints provided by the CodeSource are processed, but the binary is not parsed when this constructor returns.

The `defensiveMode` parameter optionally trades off coverage for safety; this mode is not recommended for most applications as it makes very conservative assumptions about control flow transfer instructions (see Section B.

`void parse()`

Recursively parses the binary represented by this CodeObject from all known function entry points (i.e., the hints provided by the CodeSource). This method and the following parsing methods may safely be invoked repeatedly if new information about function locations is provided through the CodeSource.

`void parse(Address target, bool recursive)`

Parses the binary starting with the instruction at the provided target address. If `recursive` is TRUE, recursive traversal parsing is used as in the default `parse()` method; otherwise only instructions reachable through intraprocedural control flow are visited.

`void parseGaps(CodeRegion *cr)`

Speculatively parse the indicated region of the binary using pattern matching to find likely function entry points. Only enabled on the x86 32-bit platform.

`Function * findFuncByEntry(CodeRegion * cr, Address entry)`

Find the function starting at address `entry` in the indicated CodeRegion. Returns NULL if no such function exists.

`int findFuncs(CodeRegion * cr, Address addr, std::set<Function*> & funcs)`

Finds all functions spanning `addr` in the code region, adding each to `funcs`. The number of results of this stabbing query are returned.

`int findFuncs(CodeRegion * cr, Address start, Address end, std::set<Function*> & funcs)`

Finds all functions overlapping the range `[start,end)` in the code region, adding each to `funcs`. The number of results of this stabbing query are returned.

`funclist & funcs()`

Returns a reference to a container of all functions in the binary. Refer to Section 4.10 for container access details.

```
Block * findBlockByEntry(CodeRegion * cr, Address entry)
```

Find the basic block starting at address `entry`. Returns NULL if no such block exists.

```
int findBlocks(CodeRegion * cr, Address addr, std::set<Block*> & blocks)
```

Finds all blocks spanning `addr` in the code region, adding each to `blocks`. Multiple blocks can be returned only on platforms with variable-length instruction sets (such as IA32) for which overlapping instructions are possible; at most one block will be returned on all other platforms.

```
void finalize()
```

Force complete parsing of the CodeObject; parsing operations are otherwise completed only as needed to answer queries.

```
CodeSource * cs()
```

Return a reference to the underlying CodeSource.

```
CFGFactory * fact()
```

Return a reference to the CFG object factory.

4.2 Class CodeSource

The CodeSource interface is used by the ParseAPI to retrieve binary code from an executable, library, or other binary code object; it also can provide hints of function entry points (such as those derived from debugging symbols) to seed the parser. The ParseAPI provides a default implementation based on the SymtabAPI that supports many common binary formats. For details on implementing a custom CodeSource, see Appendix A.

```
virtual bool nonReturning(Address func_entry)
virtual bool nonReturning(std::string func_name)
```

Looks up whether a function returns (by name or location). This information may be statically known for some code sources, and can lead to better parsing accuracy.

```
virtual Address baseAddress()
virtual Address loadAddress()
```

If the binary file type supplies non-zero base or load addresses (e.g. Windows PE), implementations should override these functions.

```
std::map< Address, std::string > & linkage()
```

Returns a reference to the external linkage map, which may or may not be filled in for a particular CodeSource implementation.

```
std::vector<CodeRegion *> const& regions()
```

Returns a read-only vector of code regions within the binary represented by this code source.

```
int findRegions(Address addr, set<CodeRegion *> & ret)
```

Finds all CodeRegion objects that overlap the provided address. Some code sources (e.g. archive files) may have several regions with overlapping address ranges; others (e.g. ELF binaries) do not.

```
bool regionsOverlap()
```

Indicates whether the CodeSource contains overlapping regions.

4.3 Class CodeRegion

The CodeRegion interface is an accounting structure used to divide CodeSources into distinct regions. This interface is mostly of interest to CodeSource implementors.

```
void names(Address addr, vector<std::string> & names)
```

Fills the provided vector with any names associated with the function at a given address in the region, e.g. symbol names in an ELF or PE binary.

`virtual bool findCatchBlock(Address addr, Address & catchStart)`

Finds the exception handler associated with an address, if one exists. This routine is only implemented for binary code sources that support structured exception handling, such as the SymtabAPI-based SymtabCodeSource provided as part of the ParseAPI.

`Address low()`

The lower bound of the interval of address space covered by this region.

`Address high()`

The upper bound of the interval of address space covered by this region.

`bool contains(Address addr)`

Returns TRUE if `addr` $\in$ [`low()`, `high()`), FALSE otherwise.

4.4 Class Function

The Function class represents the protion of the program CFG that is reachable through intraprocedural control flow transfers from the function's entry block. Functions in the ParseAPI have only a single entry point; multiple-entry functions such as those found in Fortran programs are represented as several functions that "share" a subset of the CFG.

FuncSource	Meaning
RT	recursive traversal (default)
HINT	specified in CodeSource hints
GAP	speculative parsing heuristics
GAPRT	recursive traversal from speculative parse
ONDEMAND	dynamically discovered at runtime

Return type of function `src()` see description below.

FuncReturnStatus	Meaning
UNSET	unparsed function (default)
NORETURN	will not return
UNKNOWN	cannot be determined statically
RETURN	may return

Return type of function `retstatus()`, see description below.

```
typedef ContainerWrapper<vector<Block*>,Block*,Block*> blocklist
typedef ContainerWrapper<vector<Edge*>,Edge*,Edge*> edgelist
```

Containers for block and edge access. Refer to Section 4.10 for details on `ContainerWrapper`. Library users *must not* rely on the underlying container type of ContainerWrapper lists, as it is subject to change.

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

Returns the name of this function.

```
CodeRegion * region()
```

Returns the CodeRegion that contains this function's entry point (functions can span multiple regions).

```
InstructionSource * isrc()
```

Returns the InstructionSource for this function.

```
CodeObject * obj()
```

Returns the CodeObject containing this function.

```
FuncSource src()
```

Returns the type of hint that identified this function's entry point.

```
FuncReturnStatus retstatus()
```

Returns the best-effort determination of whether this function may return or not. Return status cannot always be statically determined, and at most can guarantee that a function *may* return, not that it *will* return.

```
Block * entry()
```

Returns the basic block at this function's entry point.

`bool parsed()`

Indicates whether this function has been parsed.

`blocklist & blocks()`

Returns a list of the basic blocks comprised by this function. The blocks are guaranteed to be sorted by starting address.

`bool contains(Block *b)`

Returns TRUE if the block is contained in this function.

`edgelist & callEdges()`

Returns a list of all outgoing call edges from this function.

`blocklist & returnBlocks()`

Returns a list of all blocks ending in a `return` instruction.

`std::vector<FuncExtent *> const& extents()`

Returns a list of contiguous extents of binary code within the function.

[The following methods provide additional details about functions to support instrumentation applications and are probably of no interest to most users.]

`bool hasNoStackFrame()`

Indicates whether the function sets up a stack frame.

`bool savesFramePointer()`

Indicates whether the function saves a stack frame pointer.

`bool cleansOwnStack()`

Indicates whether the function tears down its own stack on return.

4.5 Class FuncExtent

Function Extents are used internally for accounting and lookup purposes. They may be useful for users who wish to precisely identify the ranges of the address space spanned by functions (functions are often discontiguous, particularly on architectures with variable length instruction sets).

`Address start()`

The start of this contiguous code extent.

`Address end()`

The end of this contiguous code extent (exclusive).

4.6 Class Block

A Block represents a basic block as defined in Section 2, and is the lowest level representation of code in the CFG.

`typedef ContainerWrapper<vector<Edge*>,Edge*,Edge*> edgelist`

Container for edge access. Refer to Section 4.10 for details. Library users *must not* rely on the underlying container type of ContainerWrapper lists, as it is subject to change.

`Address start()`

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

`Address end()`

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

`Address lastInsnAddr()`

Returns the address of the last instruction in this block.

`Address size()`

Returns `end()` − `start()`.

`bool parsed()`

Indicates whether this block has been parsed.

`CodeObject * obj()`

Returns the CodeObject containing this block.

`CodeRegion * region()`

Returns the CodeRegion containing this block.

`edgelist & sources()`

Return a list of all incomming edges to the block.

`edgelist & targets()`

Return a list of all outgoing edges from the block.

`bool consistent(Address addr, Address & prev_insn)`

Check whether address `addr` is *consistent* with this basic block. An address is consistent if it is the boundary between two instructions in the block. As long as `addr` is within the range of the block, `prev_insn` will contain the address of the previous instruction boundary before `addr`, regardless of whether `addr` is consistent or not.

`int containingFuncs()`

Returns the number of functions that contain this block.

`void getFuncs(std::vector<Function *> & funcs)`

Fills in the provided vector with all functions that share this basic block.

4.7 Class Edge

Typed Edges join two blocks in the CFG, indicating the type of control flow transfer instruction that joins the blocks to each other. Edges may not correspond to a control flow transfer instruction at all, as in the case of the FALLTHROUGH edge that indicates where straight-line control flow is split by incoming transfers from another location, such as a branch. While not all blocks end in a control transfer instruction, all control transfer instructions end basic blocks and have outgoing edges; in the case of unresolvable control flow, the edge will target a special "sink" block (see `sinkEdge()`, below.

EdgeTypeEnum	Meaning
CALL	call edge
COND_TAKEN	conditional branch–taken
COND_NOT_TAKEN	conditional branch–not taken
INDIRECT	branch indirect
DIRECT	branch direct
FALLTHROUGH	direct fallthrough (no branch)
CATCH	exception handler
CALL_FT	post-call fallthrough
RET	return

`Block * src()`

Returns the source block of this edge.

`Block * trg()`

Returns the target block of this edge.

`EdgeTypeEnum type()`

Returns the edge type.

`bool sinkEdge()`

Indicates whether this edge targets the special *sink* block.

`bool interproc()`

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

4.8 Class EdgePredicate

Edge predicates control iteration over edges. For example, the provided `Intraproc` edge predicate can be passed to an edge iterator constructor, ensuring that only intraprocedural edges are visited during iteration. Two other implementations of EdgePredicate are provided: `SingleContext` only visits edges that stay in a single function context, and `NoSinkPredicate` does not visit edges to the *sink* block. The following code traverses all of the basic blocks within a function:

```
vector<Block*> work;
std::map<Block*,bool> seen; // avoid loops
Intraproc epred; // ignore calls, returns

work.push_back(func->entry()); // assuming 'func' is a Function*
seen[func->entry()] = true;
while(!work.empty()) {
    Block * b = work.back();
    work.pop_back();

    // do some stuff with b...

    Block::edgelist & targets = block->targets();
    Block::edgelist::iterator eit = targets.begin(&epred);
    for( ; eit != targets.end(); ++eit) {
        Edge * e = (*eit);
        if(seen.find(e->trg()) == seen.end())
            work.push_back(e->trg());
    }
}
```

New edge predicates can be created by implementing the following simple interface:

```
EdgePredicate()
EdgePredicate(EdgePredicate * next)
```

Constructs a predicate, either with or without a previously existing predicate to chain it to. Chained predicates return the logical AND over all predicates in the chain.

```
virtual bool pred_impl(Edge *)
```

Evaluates the predicate on the provided edge.

4.9 Class ParseCallback

The ParseCallback class allows ParseAPI users to be notified of various events during parsing. For most users this notification is unnecessary, and an instantiation of the default ParseCallback can

be passed to the CodeObject during initialization. Users who wish to be notified must implement a class that inherits from ParseCallback, and implement one or more of the methods described below to receive notification of those events.

```
struct default_details {
    default_details(unsigned char*b,size_t s, bool ib) : ibuf(b), isize(s), isbranch(ib) { }
    unsigned char * ibuf;
    size_t isize;
    bool isbranch;
}
```

Details used in the `unresolved_cf` and `abruptEnd_cf` callbacks.

```
virtual void instruction_cb(Function*,Address,insn_details*)
```

Invoked for each instruction decoded during parsing. Implementing this callback may incur significant overhead.

```
struct insn_details {
    InsnAdapter::InstructionAdapter * insn;
}
```

```
void interproc_cf(Function*,Address,interproc_details*)
```

Invoked for each interprocedural control flow instruction.

```
struct interproc_details {
    typedef enum {
        ret,
        call,
        branch_interproc, // tail calls, branches to plts
        syscall
    } type_t;
    unsigned char * ibuf;
    size_t isize;
    type_t type;
    union {
        struct {
            Address target;
            bool absolute_address;
            bool dynamic_call;
        } call;
    } data;
}
```

Details used in the `interproc_cf` callback.

```
void overlapping_blocks(Block*,Block*)
```

Noification of inconsistent parse data (overlapping blocks).

4.10 Containers

Several of the ParseAPI data structures export containers of CFG objects; the CodeObject provides a list of functions in the binary, for example, while functions provide lists of blocks and so on. To avoid tying the internal storage for these structures to any particular container type, ParseAPI objects export a ContainerWrapper that provides an iterator interface to the internal containers. These wrappers and predicate interfaces are designed to add minimal overhead while protecting ParseAPI users from exposure to internal container storage details. Users *must not* rely on properties of the underlying container type (e.g. storage order) unless that property is explicity stated in this manual.

ContainerWrapper containers export the following interface (`iterator` types vary depending on the template parameters of the ContainerWrapper, but are always instantiations of the PredicateIterator described below):

```
iterator begin()
iterator begin(predicate *)
```

Return an iterator pointing to the beginning of the container, with or without a filtering predicate implementation (see Section 4.8 for details on filter predicates).

```
iterator const& end()
```

Return the iterator pointing to the end of the container (past the last element).

```
size_t size()
```

Returns the number of elements in the container. Execution cost may vary depending on the underlying container type.

```
bool empty()
```

Indicates whether the container is empty or not.

The elements in ParseAPI containers can be accessed by iteration using an instantiation of the PredicateIterator. These iterators can optionally act as filters, evaluating a boolean predicate for each element and only returning those elements for which the predicate returns TRUE. *Iterators with non-*NULL *predicates may return fewer elements during iteration than their* `size()` *method indicates.* Currently PredicateIterators only support forward iteration. The operators `++` (prefix and postfix), `==`, `!=`, and `*` (dereference) are supported.

A Extending ParseAPI

The ParseAPI is design to be a low level toolkit for binary analysis tools. Users can extend the ParseAPI in two ways: by extending the control flow structures (Functions, Blocks, and Edges) to incorporate additional data to support various analysis applications, and by adding additional binary code sources that are unsupported by the default SymtabAPI-based code source. For example, a code source that represents a program image in memory could be implemented by fulfilling the CodeSource and InstructionSource interfaces described in Section 4.2 and below. Implementations that extend the CFG structures need only provide a custom allocation factory in order for these objects to be allocated during parsing.

A.1 Instruction and Code Sources

A CodeSource, as described above, exports its own and the InstructionSource interface for access to binary code and other details. In addition to implementing the virtual methods in the CodeSource base class (Section 4.2), the methods in the pure-virtual InstructionSource class must be implemented:

```
virtual bool isValidAddress(const Address)
```

Returns TRUE if the address is a valid code location.

```
virtual void* getPtrToInstruction(const Address)
```

Returns pointer to raw memory in the binary at the provided address.

```
virtual void* getPtrToData(const Address)
```

Returns pointer to raw memory in the binary at the provided address. The address need not correspond to an executable code region.

```
virtual unsigned int getAddressWidth()
```

Returns the address width (e.g. four or eight bytes) for the represented binary.

`virtual bool isCode(const Address)`

Indicates whether the location is in a code region.

`virtual bool isData(const Address)`

Indicates whether the location is in a data region.

`virtual Address offset()`

The start of the region covered by this instruction source.

`virtual Address length()`

The size of the region.

`virtual Architecture getArch()`

The architecture of the instruction source. See the Dyninst manual for details on architecture differences.

`virtual bool isAligned(const Address)`

For fixed-width instruction architectures, must return TRUE if the address is a valid instruction boundary and FALSE otherwise; otherwise returns TRUE. This method has a default implementation that should be sufficient.

CodeSource implementors need to fill in several data structures in the base CodeSource class:

`std::map<Address, std::string> _linkage`

Entries in the linkage map represent external linkage, e.g. the PLT in ELF binaries. Filling in this map is optional.

`Address _table_of_contents`

Many binary format have "table of contents" structures for position independant references. If such a structure exists, its address should be filled in.

```
std::vector<CodeRegion *> _regions
Dyninst::IBSTree<CodeRegion> _region_tree
```

One or more contiguous regions of code or data in the binary object must be registered with the base class. Keeping these structures in sync is the responsibility of the implementing class.

```
std::vector<Hint> _hints
```

CodeSource implementors can supply a set of Hint objects describing where functions are known to start in the binary. These hints are used to seed the parsing algorithm. Refer to the CodeSource header file for implementation details.

A.2 CFG Object Factories

Users who which to incorporate the ParseAPI into large projects may need to store additional information about CFG objects like Functions, Blocks, and Edges. The simplest way to associate the ParseAPI-level CFG representation with higher-level implementation is to extend the CFG classes provided as part of the ParseAPI. Because the parser itself does not know how to construct such extended types, implementors must provide an implementation of the CFGFactory that is specialized for their CFG classes. The CFGFactory exports the following simple interface:

```
virtual Function * mkfunc(Address addr,
    FuncSource src,
    std::string name,
    CodeObject * obj,
    CodeRegion * region,
    Dyninst::InstructionSource * isrc)
```

Returns an object derived from Function as though the provided parameters had been passed to the Function constructor. The ParseAPI parser will never invoke `mkfunc()` twice with identical `addr`, and `region` parameters—that is, Functions are guaranteed to be unique by address within a region.

```
virtual Block * mkblock(Function * func, CodeRegion * region, Address addr)
```

Returns an object derived from Block as though the provided parameters had been passed to the Block constructor. The parser will never invoke `mkblock()` with identical `addr` and `region` parameters.

```
virtual Edge * mkedge(Block * src, Block * trg, EdgeTypeEnum type)
```

Returns an object derived from Edge as though the provided parameters had been passed to the Edge constructor. The parser *may* invoke `mkedge()` multiple times with identical parameters.

```
virtual Block * mksink(CodeObject *obj, CodeRegion *r)
```

Returns a "sink" block derived from Block to which all unresolvable control flow instructions will be linked. Implementors may return a unique sink block per CodeObject or a single global sink.

Implementors of extended CFG classes are required to override the default implementations of the *mk** functions to allocate and return the appropriate derived types statically cast to the base type. Implementors must also add all allocated objects to the following internal lists:

```
fact_list<Edge> edges_
fact_list<Block> blocks_
fact_list<Function> funcs_
```

O(1) allocation lists for CFG types. See the CFG.h header file for list insertion and removal operations.

Implementors *may* but are *not required to* override the deallocation following deallocation routines. The primary reason to override these routines is if additional action or cleanup is necessary upon CFG object release; the default routines simply remove the objects from the allocation list and invoke their destructors.

```
virtual void free_func(Function * f)
virtual void free_block(Block * b)
virtual void free_edge(Edge * e)
virtual void free_all()
```

CFG objects should be freed using these functions, rather than delete, to avoid leaking memory.

B Defensive Mode Parsing

Binary code that defends itself against analysis may violate the assumptions made by the the ParseAPI's standard parsing algorithm. Enabling defensive mode parsing activates more conservative assumptions that substantially reduce the percentage of code that is analyzed by the ParseAPI. For this reason, defensive mode parsing is best-suited for use of ParseAPI in conjunction with dynamic analysis techniques that can compensate for its limited coverage of the binary code. This mode of parsing will be brought to full functionality in an upcoming release.