cle

The angr Project

Jun 04, 2026

CONTENTS:

This documentation has two sections. The first shows basic usage of CLE, and the second shows the API reference.

CHAPTER

ONE

CLE

CLE loads binaries and their associated libraries, resolves imports and provides an abstraction of process memory the same way as if it was loader by the OS's loader.

1.1 Project Links

Project repository: https://github.com/angr/cle

Documentation: https://api.angr.io/projects/cle/en/latest/

1.2 Installation

```
pip install cle
```

1.3 Usage example

```
>>> import cle
>>> ld = cle.Loader("/bin/ls")
>>> hex(ld.main_object.entry)
'0x4048d0'
>>> ld.shared_objects
{'ld-linux-x86-64.so.2': <ELF Object ld-2.21.so, maps [0x5000000:0x522312f]>,
 'libacl.so.1': <ELF Object libacl.so.1.1.0, maps [0x2000000:0x220829f]>,
 'libattr.so.1': <ELF Object libattr.so.1.1.0, maps [0x4000000:0x4204177]>,
 'libc.so.6': <ELF Object libc-2.21.so, maps [0x3000000:0x33a1a0f]>,
 'libcap.so.2': <ELF Object libcap.so.2.24, maps [0x1000000:0x1203c37]>}
>>> ld.addr_belongs_to_object(0x5000000)
<ELF Object ld-2.21.so, maps [0x5000000:0x522312f]>
>>> libc_main_reloc = ld.main_object.imports['__libc_start_main']
>>> hex(libc_main_reloc.addr)       # Address of GOT entry for libc_start_main
'0x61c1c0'
>>> import pyvex
>>> some_text_data = ld.memory.load(ld.main_object.entry, 0x100)
>>> irsb = pyvex.lift(some_text_data, ld.main_object.entry, ld.main_object.arch)
>>> irsb.pp()
IRSB {
   t0:Ity_I32 t1:Ity_I32 t2:Ity_I32 t3:Ity_I64 t4:Ity_I64 t5:Ity_I64 t6:Ity_I32 t7:Ity_
```

(continues on next page)

(continued from previous page)

```
↪I64 t8:Ity_I32 t9:Ity_I64 t10:Ity_I64 t11:Ity_I64 t12:Ity_I64 t13:Ity_I64 t14:Ity_I64

   15 | ------ IMark(0x4048d0, 2, 0) ------
   16 | t5 = 32Uto64(0x00000000)
   17 | PUT(rbp) = t5
   18 | t7 = GET:I64(rbp)
   19 | t6 = 64to32(t7)
   20 | t2 = t6
   21 | t9 = GET:I64(rbp)
   22 | t8 = 64to32(t9)
   23 | t1 = t8
   24 | t0 = Xor32(t2,t1)
   25 | PUT(cc_op) = 0x0000000000000013
   26 | t10 = 32Uto64(t0)
   27 | PUT(cc_dep1) = t10
   28 | PUT(cc_dep2) = 0x0000000000000000
   29 | t11 = 32Uto64(t0)
   30 | PUT(rbp) = t11
   31 | PUT(rip) = 0x00000000004048d2
   32 | ------ IMark(0x4048d2, 3, 0) ------
   33 | t12 = GET:I64(rdx)
   34 | PUT(r9) = t12
   35 | PUT(rip) = 0x00000000004048d5
   36 | ------ IMark(0x4048d5, 1, 0) ------
   37 | t4 = GET:I64(rsp)
   38 | t3 = LDle:I64(t4)
   39 | t13 = Add64(t4,0x0000000000000008)
   40 | PUT(rsp) = t13
   41 | PUT(rsi) = t3
   42 | PUT(rip) = 0x00000000004048d6
   43 | t14 = GET:I64(rip)
   NEXT: PUT(rip) = t14; Ijk_Boring
}
```

1.4 Valid options

For a full listing and description of the options that can be provided to the loader and the methods it provides, please examine the docstrings in `cle/loader.py`. If anything is unclear or poorly documented (there is much) please complain through whatever channel you feel appropriate.

1.5 Loading Backends

CLE's loader is implemented in the Loader class. There are several backends that can be used to load a single file:

- ELF, as its name says, loads ELF binaries. ELF files loaded this way are statically parsed using PyElfTools.
- PE is a backend to load Microsoft's Portable Executable format, effectively Windows binaries. It uses the (optional) `pefile` module.
- Mach-O is a backend to load, you guessed it, Mach-O binaries. Support is limited for this backend.
- Blob is a backend to load unknown data. It requires that you specify the architecture it would be run on, in the form of a class from ArchInfo.

Which backend you use can be specified as an argument to Loader. If left unspecified, the loader will pick a reasonable default.

1.6 Finding shared libraries

- If the `auto_load_libs` option is set to False, the Loader will not automatically load libraries requested by loaded objects. Otherwise…
- The loader determines which shared objects are needed when loading binaries, and searches for them in the following order:
  - in the current working directory
  - in folders specified in the `ld_path` option
  - in the same folder as the main binary
  - in the system (in the corresponding library path for the architecture of the binary, e.g., /usr/arm-linux-gnueabi/lib for ARM, note that you need to install cross libraries for this, e.g., libc6-powerpc-cross on Debian - needs emdebian repos)
  - in the system, but with mismatched version numbers from what is specified as a dependency, if the `ignore_import_version_numbers` option is True
- If no binary is found with the correct architecture, the loader raises an exception if `except_missing_libs` option is True. Otherwise it simply leaves the dependencies unresolved.

CHAPTER

TWO

API DOCUMENTATION

2.1 Loading Interface

class cle.**Loader**

Bases: object

The loader loads all the objects and exports an abstraction of the memory of the process. What you see here is an address space with loaded and rebased binaries.

__init__(*main_binary: str | BinaryIO | Path | Backend, auto_load_libs: bool = False, concrete_target=None, force_load_libs: Iterable[str | BinaryIO | Path] = (), skip_libs: Iterable[str] = (), main_opts: dict[str, Any] | None = None, lib_opts: dict[str, dict[str, Any]] | None = None, ld_path: Iterable[str | Path] = (), use_system_libs: bool = True, ignore_import_version_numbers: bool = True, case_insensitive: bool = False, rebase_granularity: int = 1048576, except_missing_libs: bool = False, aslr: bool = False, perform_relocations: bool = True, load_debug_info: bool = False, page_size: int = 1, preload_libs: Iterable[str | BinaryIO | Path] = (), arch: Arch | str | None = None, force_tls: str | None = None*)

Parameters

- **main_binary** (str | BinaryIO | Path | *Backend*) – The path to the main binary you're loading, or a file-like object with the binary in it.
- **auto_load_libs** (bool) – Whether to automatically load shared libraries that loaded objects depend on.
- **load_debug_info** (bool) – Whether to automatically parse DWARF data and search for debug symbol files.
- **concrete_target** – Whether to instantiate a concrete target for a concrete execution of the process. if this is the case we will need to instantiate a SimConcreteEngine that wraps the ConcreteTarget provided by the user.
- **force_load_libs** (Iterable[str | BinaryIO | Path]) – A list of libraries to load regardless of if they're required by a loaded object.
- **skip_libs** (Iterable[str]) – A list of libraries to never load, even if they're required by a loaded object.
- **main_opts** (dict[str, Any] | None) – A dictionary of options to be used loading the main binary.
- **lib_opts** (dict[str, dict[str, Any]] | None) – A dictionary mapping library names to the dictionaries of options to be used when loading them.
- **ld_path** (Iterable[str | Path]) – A list of paths in which we can search for shared libraries.

- **use_system_libs** (bool) – Whether or not to search the system load path for requested libraries. Default True.
- **ignore_import_version_numbers** (bool) – Whether libraries with different version numbers in the filename will be considered equivalent, for example libc.so.6 and libc.so.0
- **case_insensitive** (bool) – If this is set to True, filesystem loads will be done case-insensitively regardless of the case-sensitivity of the underlying filesystem.
- **rebase_granularity** (int) – The alignment to use for rebasing shared objects
- **except_missing_libs** (bool) – Throw an exception when a shared library can't be found.
- **aslr** (bool) – Load libraries in symbolic address space. Do not use this option.
- **page_size** (int) – The granularity with which data is mapped into memory. Set to 0x1000 if you are working in an environment where data will always be memory mapped in a page-graunlar way.
- **preload_libs** (Iterable[str | BinaryIO | Path]) – Similar to *force_load_libs* but will provide for symbol resolution, with precedence over any dependencies.
- **perform_relocations** (*bool*)
- **arch** (*Arch | str | None*)
- **force_tls** (*str | None*)

Variables

- **memory** (cle.memory.Clemory) – The loaded, rebased, and relocated memory of the program.
- **main_object** – The object representing the main binary (i.e., the executable).
- **shared_objects** – A dictionary mapping loaded library names to the objects representing them.
- **all_objects** – A list containing representations of all the different objects loaded.
- **requested_names** – A set containing the names of all the different shared libraries that were marked as a dependency by somebody.
- **initial_load_objects** – A list of all the objects that were loaded as a result of the initial load request.

When reference is made to a dictionary of options, it requires a dictionary with zero or more of the following keys:

- backend : "elf", "pe", "mach-o", "blob" : which loader backend to use
- arch : The archinfo.Arch object to use for the binary
- base_addr : The address to rebase the object at
- entry_point : The entry point to use for the object

More keys are defined on a per-backend basis.

property main_object: *Backend*

property original_main_object: *Backend*

property memory: *Clemory*

property memory_ro_view: ClemoryReadOnlyView | None

property tls: *ThreadManager*

close()

property max_addr: int

The maximum address loaded as part of any loaded object (i.e., the whole address space).

property min_addr: int

The minimum address loaded as part of any loaded object (i.e., the whole address space).

property initializers: list[int]

Return a list of all the initializers that should be run before execution reaches the entry point, in the order they should be run.

property finalizers: list[int]

Return a list of all the finalizers that should be run before the program exits. I'm not sure what order they should be run in.

property linux_loader_object: *Backend* | None

If the linux dynamic loader is present in memory, return it

property elfcore_object: *ELFCore* | None

If a corefile was loaded, this returns the actual core object instead of the main binary

property extern_object: *ExternObject*

Return the extern object used to provide addresses to unresolved symbols and angr internals.

Accessing this property will load this object into memory if it was not previously present.

proposed model for how multiple extern objects should work:

1) extern objects are a linked list. the one in loader._extern_object is the head of the list
2) **each round of explicit loads generates a new extern object if it has unresolved dependencies. this object**
   has exactly the size necessary to hold all its exports.
3) **All requests for size are passed down the chain until they reach an object which has the space to service**
   it or an object which has not yet been mapped. If all objects have been mapped and are full, a new extern object is mapped with a fixed size.

property kernel_object: *KernelObject*

Return the object used to provide addresses to syscalls.

Accessing this property will load this object into memory if it was not previously present.

property all_elf_objects: list[*MetaELF*]

Return a list of every object that was loaded from an ELF file.

property all_pe_objects: list[*PE*]

Return a list of every object that was loaded from an ELF file.

property missing_dependencies: set[str]

Return a set of every name that was requested as a shared object dependency but could not be loaded

property auto_load_libs: bool

describe_addr(*addr: int*) → str

Returns a textual description of what's in memory at the provided address

Return type
str

Parameters
addr (*int*)

find_object(*spec:* Backend | *str*, *extra_objects: Iterable[*Backend*] = ()*) → *Backend* | None

If the given library specification has been loaded, return its object, otherwise return None.

Return type
Backend | None

Parameters

- **spec** (Backend | *str*)
- **extra_objects** (*Iterable[*Backend*]*)

find_object_containing(*addr: int*, *membership_check: bool = True*) → *Backend* | None

Return the object that contains the given address, or None if the address is unmapped. This lookup method does not include outer objects (obj.is_outer is True).

Parameters

- **addr** (int) – The address that should be contained in the object.
- **membership_check** (bool) – Whether a membership check should be performed or not (True by default). This option can be set to False if you are certain that the target object does not have "holes".

Return type
Backend | None

Returns
The object or None.

find_segment_containing(*addr: int*, *skip_pseudo_objects: bool = True*) → *Segment* | None

Find the section object that the address belongs to.

Parameters

- **addr** (int) – The address to test
- **skip_pseudo_objects** (bool) – Skip objects that CLE adds during loading.

Returns
The section that the address belongs to, or None if the address does not belong to any section, or if section information is not available.

Return type
Segment | None

find_section_containing(*addr: int*, *skip_pseudo_objects=True*) → *Section* | None

Find the section object that the address belongs to.

Parameters

- **addr** (int) – The address to test.
- **skip_pseudo_objects** (*bool*) – Skip objects that CLE adds during loading.

Returns
The section that the address belongs to, or None if the address does not belong to any section, or if section information is not available.

Return type
Section | None

find_loadable_containing(*addr: int*, *skip_pseudo_objects=True*) → *Region* | None

Find the section or segment object the address belongs to. Sections will only be used if the corresponding object does not have segments.

Parameters

- **addr** (int) – The address to test
- **skip_pseudo_objects** – Skip objects that CLE adds during loading.

Return type
Region | None

Returns
The section or segment that the address belongs to, or None if the address does not belong to any section or segment.

find_section_next_to(*addr: int*, *skip_pseudo_objects=True*) → *Section* | None

Find the next section after the given address.

Parameters

- **addr** (int) – The address to test.
- **skip_pseudo_objects** (*bool*) – Skip objects that CLE adds during loading.

Returns
The next section that goes after the given address, or None if there is no section after the address, or if section information is not available.

Return type
Section | None

find_symbol(*thing*, *fuzzy=False*) → *Symbol* | None

Search for the symbol with the given name or address.

Parameters

- **thing** – Either the name or address of a symbol to look up
- **fuzzy** – Set to True to return the first symbol before or at the given address

Return type
Symbol | None

Returns
A `cle.backends.Symbol` object if found, None otherwise.

property symbols: Iterator[*Symbol*]

find_all_symbols(*name: str*, *exclude_imports=True*, *exclude_externs=False*, *exclude_forwards=True*) → Iterable[*Symbol*]

Iterate over all symbols present in the set of loaded binaries that have the given name

Parameters

- **name** (str) – The name to search for

- **exclude_imports** – Whether to exclude import symbols. Default True.
- **exclude_externs** – Whether to exclude symbols in the extern object. Default False.
- **exclude_forwards** – Whether to exclude forward symbols. Default True.

Return type
Iterable[*Symbol*]

find_plt_stub_name(*addr: int*) → str | None

Return the name of the PLT stub starting at addr.

Return type
str | None

Parameters
addr (*int*)

find_relevant_relocations(*name: str*) → Iterator[*Relocation*]

Iterate through all the relocations referring to the symbol with the given name

Return type
Iterator[*Relocation*]

Parameters
name (*str*)

perform_irelative_relocs(*resolver_func*)

Use this method to satisfy IRelative relocations in the binary that require execution of loaded code.

Note that this does NOT handle IFunc symbols, which must be handled separately. (this could be changed, but at the moment it's desirable to support lazy IFunc resolution, since emulation is usually slow)

Parameters
resolver_func – A callback function that takes an address, runs the code at that address, and returns the return value from the emulated function.

dynamic_load(*spec*)

Load a file into the address space. Note that the sematics of auto_load_libs and except_missing_libs apply at all times.

Parameters
spec – The path to the file to load. May be an absolute path, a relative path, or a name to search in the load path.

Returns
A list of all the objects successfully loaded, which may be empty if this object was previously loaded. If the object specified in spec failed to load for any reason, including the file not being found, return None.

get_loader_symbolic_constraints()

Do not use this method.

gen_ro_memview() → None

Generate a read-only view of the memory, and update self._memory_ro_view for faster data loading. Please call this method again for updating the read-only view, or discard_ro_memview() to discard any previously generated read-only views.

Return type
None

discard_ro_memview() → None

Discard any previously generated read-only views of the memory.

Return type
None

fast_memory_load_pointer(*addr: int*, *size: int | None = None*) → int | None

Perform a fast memory loading of a pointer.

Parameters

- **addr** (int) – Address to read from.
- **size** (int | None) – Size of the pointer. Default to machine-word size.

Return type
int | None

Returns
A pointer or None if the address does not exist.

2.2 Backend Interface

class cle.backends.backend.**FunctionHintSource**

Bases: object

Enums that describe the source of function hints.

EH_FRAME = 0

EXTERNAL_EH_FRAME = 1

EXPORT_TABLE = 2

class cle.backends.backend.**FunctionHint**

Bases: object

Describes a function hint.

Variables

- **addr** (*int*) – Address of the function.
- **size** (*int*) – Size of the function.
- **source** (*int*) – Source of this hint.
- **name** (*str | None*) – Optional symbol name, if known.

__init__(*addr*, *size*, *source*, *name=None*)

addr

size

source

name

class cle.backends.backend.**ExceptionHandling**

Bases: object

Describes an exception handling.

Exception handlers are usually language-specific. In C++, it is usually implemented as try {} catch {} blocks.

Variables

- **start_addr** (*int*) – The beginning of the try block.
- **size** (*int*) – Size of the try block.
- **handler_addr** (*Optional[int]*) – Address of the exception handler code.
- **type** – Type of the exception handler. Optional.
- **func_addr** (*Optional[int]*) – Address of the function. Optional.

__init__(*start_addr*, *size*, *handler_addr=None*, *type_=None*, *func_addr=None*)

start_addr

size

handler_addr

type

func_addr

class cle.backends.backend.**Backend**

Bases: object

Main base class for CLE binary objects.

An alternate interface to this constructor exists as the static method cle.loader.Loader.load_object()

Variables

- **binary** – The path to the file this object is loaded from
- **binary_basename** – The basename of the filepath, or a short representation of the stream it was loaded from
- **is_main_bin** – Whether this binary is loaded as the main executable
- **segments** – A listing of all the loaded segments in this file
- **sections** – A listing of all the demarked sections in the file
- **sections_map** – A dict mapping from section name to section
- **imports** (dict[str, *Relocation*]) – A mapping from symbol name to import relocation
- **resolved_imports** – A list of all the import symbols that are successfully resolved
- **relocs** (list[*Relocation*]) – A list of all the relocations in this binary
- **irelatives** – A list of tuples representing all the irelative relocations that need to be performed. The first item in the tuple is the address of the resolver function, and the second item is the address of where to write the result. The destination address is an RVA.
- **jmprel** – A mapping from symbol name to the address of its jump slot relocation, i.e. its GOT entry.
- **arch** (*archinfo.arch.Arch*) – The architecture of this binary

- **os** (*str*) – The operating system this binary is meant to run under
- **mapped_base** (*int*) – The base address of this object in virtual memory
- **deps** – A list of names of shared libraries this binary depends on
- **linking** – 'dynamic' or 'static'
- **linked_base** – The base address this object requests to be loaded at
- **pic** (*bool*) – Whether this object is position-independent
- **execstack** (*bool*) – Whether this executable has an executable stack
- **provides** (*str*) – The name of the shared library dependancy that this object resolves
- **symbols** (*list*) – A list of symbols provided by this object, sorted by address
- **has_memory** – Whether this backend is backed by a Clemory or not. As it stands now, a backend should still define *min_addr* and *max_addr* even if *has_memory* is False.

is_default = False

is_outer = False

__init__(*binary*, *binary_stream*, *loader=None*, *is_main_bin=False*, *entry_point=None*, *arch=None*, *base_addr=None*, *force_rebase=False*, *has_memory=True*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

load_args: dict[str, Any]

unpacked_name: str | None

imports: dict[str, *Relocation*]

relocs: list[*Relocation*]

child_objects: list[*Backend*]

exception_handlings: list[*ExceptionHandling*]

function_hints: list[*FunctionHint*]

meta_regions: list[MemRegion]

memory: *Clemory*

cached_content: bytes | None

property md5: bytes | None

MD5 digest of the binary, computed lazily on first access.

property sha256: bytes | None

SHA-256 digest of the binary, computed lazily on first access.

property arch: Arch

property loader: *Loader*

close() → None

Return type
None

set_arch(*arch*)

set_load_args(**kwargs*) → None

Return type
None

property image_base_delta

property entry

property segments: *Regions*[*Segment*]

property sections: *Regions*[*Section*]

property symbols_by_addr

rebase(*new_base*)

Rebase backend's regions to the new base where they were mapped by the loader

relocate()

Apply all resolved relocations to memory.

The meaning of "resolved relocations" is somewhat subtle - there is a linking step which attempts to resolve each relocation, currently only present in the main internal loading function since the calculation of which objects should be available

contains_addr(*addr*)

Is *addr* in one of the binary's segments/sections we have loaded? (i.e. is it mapped into memory ?)

find_loadable_containing(*addr*)

find_segment_containing(*addr: int*) → *Segment* | None

Returns the segment that contains *addr*, or None.

Return type
Segment | None

Parameters
addr (*int*)

find_section_containing(*addr: int*) → *Section* | None

Returns the section that contains *addr* or None.

Return type
Section | None

Parameters
addr (*int*)

addr_to_offset(*addr: int*) → int | None

Return type
int | None

Parameters
addr (*int*)

offset_to_addr(*offset: int*) → int | None

Return type
int | None

Parameters
offset (*int*)

property min_addr: int

This returns the lowest virtual address contained in any loaded segment of the binary.

property max_addr: int

This returns the highest virtual address contained in any loaded segment of the binary.

property initializers: list[int]

Stub function. Should be overridden by backends that can provide initializer functions that ought to be run before execution reaches the entry point. Addresses should be rebased.

property finalizers: list[int]

Stub function. Like initializers, but with finalizers.

property threads: list

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This property should contain a list of names for these threads, which should be unique.

thread_registers(*thread=None*) → dict[str, Any]

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This method should return the register file for a given thread (as named in `Backend.threads`) as a dict mapping register names (as seen in archinfo) to numbers. If the thread is not specified, it should return the context for a "default" thread. If there are no threads, it should return an empty dict.

Return type
dict[str, Any]

initial_register_values()

Deprecated

get_symbol(*name: str*) → *Symbol* | None

Stub function. Implement to find the symbol with name *name*.

Return type
Symbol | None

Parameters
name (*str*)

property available_arches: list[str]

Return the list of architecture names available in this binary.

For most backends this is a single-element list derived from the loaded architecture. Container backends (e.g. Universal2) override this to report every architecture present in the file.

static extract_soname(*path*) → str | None

Extracts the shared object identifier from the path, or returns None if it cannot.

Return type
str | None

classmethod is_compatible(*stream*) → bool

Determine quickly whether this backend can load an object from this stream

Return type
bool

classmethod check_compatibility(*spec*, *obj*) → bool

Performs a minimal static load of `spec` and returns whether it's compatible with other_obj

Return type
bool

classmethod check_magic_compatibility(*stream: BinaryIO*) → bool

Check if a stream of bytes contains the same magic number as the main object

Return type
bool

Parameters
stream (*BinaryIO*)

cle.backends.backend.**register_backend**(*name*, *cls*)

class cle.backends.symbol.**SymbolType**

Bases: Enum

ABI-agnostic symbol types

TYPE_OTHER = 0

TYPE_NONE = 1

TYPE_FUNCTION = 2

TYPE_OBJECT = 3

TYPE_SECTION = 4

TYPE_TLS_OBJECT = 5

TYPE_FUNCTION_OR_OBJECT = 6

class cle.backends.symbol.**SymbolSubType**

Bases: Enum

Abstract base class for ABI-specific symbol types

to_base_type() → *SymbolType*

A subclass' ABI-specific mapping to :SymbolType:

Return type
SymbolType

class cle.backends.symbol.**Symbol**

Bases: object

Representation of a symbol from a binary file. Smart enough to rebase itself.

There should never be more than one Symbol instance representing a single symbol. To make sure of this, only use the `cle.backends.Backend.get_symbol()` to create new symbols.

Variables

- **owner** (*cle.backends.Backend*) – The object that contains this symbol
- **name** (*str*) – The name of this symbol
- **addr** (*int*) – The un-based address of this symbol, an RVA
- **size** (*int*) – The size of this symbol
- **_type** (*SymbolType*) – The ABI-agnostic type of this symbol
- **resolved** (*bool*) – Whether this import symbol has been resolved to a real symbol
- **resolvedby** (*None or cle.backends.Symbol*) – The real symbol this import symbol has been resolve to
- **resolvewith** (*str*) – The name of the library we must use to resolve this symbol, or None if none is required.

__init__(*owner:* Backend, *name: str*, *relative_addr: int*, *size: int*, *sym_type:* SymbolType)

Not documenting this since if you try calling it, you're wrong.

Parameters

- **owner** (Backend)
- **name** (*str*)
- **relative_addr** (*int*)
- **size** (*int*)
- **sym_type** (SymbolType)

owner: *Backend*

resolve(*obj*)

property type: *SymbolType*

The ABI-agnostic SymbolType. Must be overridden by derived types.

property subtype: *SymbolSubType*

A subclass' ABI-specific types

property rebased_addr

The address of this symbol in the global memory space

property linked_addr

property is_function

Whether this symbol is a function

is_static = False

is_common = False

is_import = False

is_export = False

is_local = False

is_weak = False

is_extern = False

is_forward = False

resolve_forwarder()

If this symbol is a forwarding export, return the symbol the forwarding refers to, or None if it cannot be found

property owner_obj

class cle.backends.regions.Regions

Bases: Generic

A container class acting as a list of regions (sections or segments). Additionally, it keeps an sorted list of all regions that are mapped into memory to allow fast lookups.

We assume none of the regions overlap with others.

__init__(*lst: list[R] | None = None*)

Parameters
lst (*list[R] | None*)

property raw_list: list[R]

Get the internal list. Any change to it is not tracked, and therefore _sorted_list will not be updated. Therefore you probably does not want to modify the list.

Returns
The internal list container.

Return type
list

property max_addr: int | None

Get the highest address of all regions.

Returns
The highest address of all regions, or None if there is no region available.

Return type
int or None

append(*region: R*)

Append a new Region instance into the list.

Parameters
region (TypeVar(R, bound= *Region*)) – The region to append.

remove(*region: R*) → None

Remove an existing Region instance from the list.

Parameters
region (TypeVar(R, bound= *Region*)) – The region to remove.

Return type
None

find_region_containing(*addr: int*) → R | None

Find the region that contains a specific address. Returns None if none of the regions covers the address.

Parameters
addr (int) – The address.

Return type
Optional[TypeVar(R, bound= *Region*)]

Returns
The region that covers the specific address, or None if no such region is found.

find_region_next_to(*addr: int*) → R | None

Find the next region after the given address.

Parameters
addr (int) – The address to test.

Return type
Optional[TypeVar(R, bound= *Region*)]

Returns
The next region that goes after the given address, or None if there is no section after the address,

class cle.backends.region.**Region**

Bases: object

A region of memory that is mapped in the object's file.

Variables

- **offset** – The offset into the file the region starts.
- **vaddr** – The virtual address.
- **filesize** – The size of the region in the file.
- **memsize** – The size of the region when loaded into memory.

The prefix *v-* on a variable or parameter name indicates that it refers to the virtual, loaded memory space, while a corresponding variable without the *v-* refers to the flat zero-based memory of the file.

When used next to each other, *addr* and *offset* refer to virtual memory address and file offset, respectively.

__init__(*offset*, *vaddr*, *filesize*, *memsize*)

vaddr: int

memsize: int

filesize: int

contains_addr(*addr*)
Does this region contain this virtual address?

contains_offset(*offset*)
Does this region contain this offset into the file?

addr_to_offset(*addr*)
Convert a virtual memory address into a file offset

offset_to_addr(*offset*)
Convert a file offset into a virtual memory address

property max_addr
The maximum virtual address of this region

property min_addr
The minimum virtual address of this region

property max_offset
The maximum file offset of this region

min_offset()
The minimum file offset of this region

property is_readable: *bool*

property is_writable: *bool*

property is_executable: *bool*

class cle.backends.region.**Segment**
Bases: *Region*

class cle.backends.region.**EmptySegment**
Bases: *Segment*

A segment with no static content, and permissions

__init__(*vaddr*, *memsize*, *is_readable=True*, *is_writable=True*, *is_executable=False*)

property is_executable

property is_writable

property is_readable

property only_contains_uninitialized_data
Whether this section is initialized to zero after the executable is loaded.

class cle.backends.region.**Section**
Bases: *Region*

Simple representation of a loaded section.

Variables
name (*str*) – The name of the section

__init__(*name*, *offset*, *vaddr*, *size*)

Parameters

- **name** (*str*) – The name of the section
- **offset** (*int*) – The offset into the binary file this section begins
- **vaddr** (*int*) – The address in virtual memory this section begins
- **size** (*int*) – How large this section is

property is_readable
Whether this section has read permissions

property is_writable
Whether this section has write permissions

property is_executable
Whether this section has execute permissions

property only_contains_uninitialized_data

Whether this section is initialized to zero after the executable is loaded.

class cle.backends.named_region.**NamedRegion**

Bases: *Backend*

A NamedRegion represents a region of memory that has a name, a location, but no static content.

This region also has permissions; with no memory, these obviously don't do anything on their own, but they help inform any other code that relies on CLE (e.g., angr)

This can be used as a placeholder for memory that should exist in CLE's view, but for which it does not need data, like RAM, MMIO, etc

is_default = False

__init__(*name*, *start*, *end*, *is_readable=True*, *is_writable=True*, *is_executable=False*, ***kwargs*)

Create a NamedRegion.

Parameters

- **name** – The name of the region
- **start** – The start address of the region
- **end** – The end address (exclusive) of the region
- **is_readable** – Whether the region is readable
- **is_writable** – Whether the region is writable
- **is_executable** – Whether the region is executable
- **kwargs**

has_memory = False

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

property min_addr

This returns the lowest virtual address contained in any loaded segment of the binary.

property max_addr

This returns the highest virtual address contained in any loaded segment of the binary.

function_name(*addr*)

NamedRegions don't support function names.

contains_addr(*addr*)

Is *addr* in one of the binary's segments/sections we have loaded? (i.e. is it mapped into memory ?)

classmethod check_compatibility(*spec*, *obj*)

Performs a minimal static load of `spec` and returns whether it's compatible with other_obj

class cle.backends.externs.**ExternSegment**

Bases: *Segment*

__init__(*map_size*)

addr_to_offset(*addr*)

Convert a virtual memory address into a file offset

offset_to_addr(*offset*)

Convert a file offset into a virtual memory address

contains_offset(*offset*)

Does this region contain this offset into the file?

is_readable = True

is_writable = True

is_executable = True

class cle.backends.externs.**TOCRelocation**

Bases: *Relocation*

property value

class cle.backends.externs.**ExternObject**

Bases: *Backend*

__init__(*loader*, *map_size=0*, *tls_size=0*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

rebase(*new_base*)

Rebase backend's regions to the new base where they were mapped by the loader

make_extern(*name*, *size=0*, *alignment=None*, *thumb=False*, *sym_type=SymbolType.TYPE_FUNCTION*, *point_to=None*, *libname=None*) → *Symbol*

Return type
Symbol

get_pseudo_addr(*name*) → int

Return type
int

allocate(*size=1*, *alignment=8*, *thumb=False*, *tls=False*) → int

Return type
int

property max_addr

This returns the highest virtual address contained in any loaded segment of the binary.

make_import(*name*, *sym_type*)

class cle.backends.externs.**KernelObject**

Bases: *Backend*

__init__(*loader*, *map_size=32768*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

add_name(*name*, *addr*)

property max_addr

This returns the highest virtual address contained in any loaded segment of the binary.

class cle.backends.externs.**PointToPrecise**

Bases: *PointTo*

pointto_precise = None

relocations()

Maybe implement me: If you like, return a list of relocation objects to apply. To create new import symbols, use `self.owner.make_extern_import`.

class cle.backends.externs.simdata.**SimData**

Bases: *Symbol*

A SimData class is used to provide data when there is an unresolved data import symbol.

To use it, subclass this class and implement the below attributes and methods.

Variables

- **name** – The name of the symbol to provide
- **libname** – The name of the library from which the symbol originally comes (currently unused).
- **type** – The type of the symbol, usually `SymbolType.TYPE_OBJECT`.

Use the below *register* method to register SimData subclasses with CLE.

NOTE: SimData.type hides the Symbol.type instance property

name: str **= NotImplemented**

type: *SymbolType* **= NotImplemented**

libname: str **= NotImplemented**

classmethod static_size(*owner*) → int

Implement me: return the size of the symbol in bytes before it gets constructed

Parameters
owner – The ExternObject owning the symbol-to-be. Useful to get at `owner.arch`.

Return type
int

value() → bytes

Implement me: the initial value of the bytes in memory for the symbol. Should return a bytestring of the same length as static_size returned. (owner is `self.owner` now)

Return type
bytes

relocations() → list[*Relocation*]

Maybe implement me: If you like, return a list of relocation objects to apply. To create new import symbols, use `self.owner.make_extern_import`.

Return type
list[*Relocation*]

cle.backends.externs.simdata.**lookup**(*name: str*, *libname*) → *type*[*SimData*] | None

Return type
type[*SimData*] | None

Parameters
name (*str*)

cle.backends.externs.simdata.**register**(*simdata_cls:* type*[*SimData*]*)

Register the given SimData class with CLE so it may be used during loading

Parameters
simdata_cls (type*[*SimData*]*)

class cle.backends.externs.simdata.simdata.**SimData**

Bases: *Symbol*

A SimData class is used to provide data when there is an unresolved data import symbol.

To use it, subclass this class and implement the below attributes and methods.

Variables

- **name** – The name of the symbol to provide
- **libname** – The name of the library from which the symbol originally comes (currently unused).
- **type** – The type of the symbol, usually `SymbolType.TYPE_OBJECT`.

Use the below *register* method to register SimData subclasses with CLE.

NOTE: SimData.type hides the Symbol.type instance property

name: str = **NotImplemented**

type: *SymbolType* = **NotImplemented**

libname: str = **NotImplemented**

classmethod static_size(*owner*) → int

Implement me: return the size of the symbol in bytes before it gets constructed

Parameters
owner – The ExternObject owning the symbol-to-be. Useful to get at `owner.arch`.

Return type
int

value() → bytes

Implement me: the initial value of the bytes in memory for the symbol. Should return a bytestring of the same length as static_size returned. (owner is `self.owner` now)

Return type
bytes

relocations() → list[*Relocation*]

Maybe implement me: If you like, return a list of relocation objects to apply. To create new import symbols, use `self.owner.make_extern_import`.

Return type
list[*Relocation*]

cle.backends.externs.simdata.simdata.**register**(*simdata_cls:* type[SimData])

Register the given SimData class with CLE so it may be used during loading

Parameters
simdata_cls (type[SimData])

cle.backends.externs.simdata.simdata.**lookup**(*name: str*, *libname*) → *type*[*SimData*] | None

Return type
type[*SimData*] | None

Parameters
name (*str*)

class cle.backends.externs.simdata.common.**StaticData**

Bases: *SimData*

A simple SimData utility class to use when you have a SimData which should provide just a static set of bytes. To use, implement the following:

Variables

- **name** – The name of the symbol to provide.
- **libname** – The name of the library from which the symbol originally comes (currently unused).
- **data** – The bytes to provide

type: *SymbolType* = 3

data: bytes = NotImplemented

classmethod static_size(*owner*)

Implement me: return the size of the symbol in bytes before it gets constructed

Parameters
owner – The ExternObject owning the symbol-to-be. Useful to get at `owner.arch`.

value()

Implement me: the initial value of the bytes in memory for the symbol. Should return a bytestring of the same length as static_size returned. (owner is `self.owner` now)

class cle.backends.externs.simdata.common.**StaticWord**

Bases: *SimData*

A simple SimData utility class to use when you have a SimData which should provide just a static integer. To use, implement the following:

Variables

- **name** – The name of the symbol to provide.

- **libname** – The name of the library from which the symbol originally comes (currently unused).
- **word** – The value to provide
- **wordsize** – (optional) The size of the value in bytes, default the CPU wordsize

type: *SymbolType* = 3

word: int = NotImplemented

wordsize: int = None

classmethod static_size(*owner*)

Implement me: return the size of the symbol in bytes before it gets constructed

Parameters
owner – The ExternObject owning the symbol-to-be. Useful to get at `owner.arch`.

value()

Implement me: the initial value of the bytes in memory for the symbol. Should return a bytestring of the same length as static_size returned. (owner is `self.owner` now)

class cle.backends.externs.simdata.common.**PointTo**

Bases: *SimData*

A simple SimData utility class to use when you have a SimData which should provide just a pointer to some other symbol. To use, implement the following:

Variables

- **name** – The name of the symbol to provide.
- **libname** – The name of the library from which the symbol originally comes (currently unused).
- **pointto_name** – The name of the symbol to point to
- **pointto_type** – The type of the symbol to point to (usually `SymbolType.TYPE_FUNCTION` or `SymbolType.TYPE_OBJECT`)
- **addend** – (optional) an integer to be added to the symbol's address before storage

pointto_name: str = NotImplemented

pointto_type: *SymbolType* = NotImplemented

type: *SymbolType* = 3

addend: int = 0

classmethod static_size(*owner*)

Implement me: return the size of the symbol in bytes before it gets constructed

Parameters
owner – The ExternObject owning the symbol-to-be. Useful to get at `owner.arch`.

value()

Implement me: the initial value of the bytes in memory for the symbol. Should return a bytestring of the same length as static_size returned. (owner is `self.owner` now)

relocations()

Maybe implement me: If you like, return a list of relocation objects to apply. To create new import symbols, use `self.owner.make_extern_import`.

class cle.backends.externs.simdata.common.**SimDataSimpleRelocation**

Bases: *Relocation*

A relocation used to implement PointTo. Pretty simple.

__init__(*owner*, *symbol*, *addr*, *addend*, *preresolved=False*)

resolve_symbol(*solist*, ***kwargs*)

property value

2.3 Backends

2.3.1 ELF Backend

class cle.backends.**ELF**

Bases: *MetaELF*

The main loader class for statically loading ELF executables. Uses the pyreadelf library where useful.

Useful backend options:

- `debug_symbols`: Provides the path to a separate file which contains the binary's debug symbols
- `discard_section_headers`: Do not parse section headers. Use this if they are corrupted or malicious.
- **discard_program_headers: Do not parse program headers. Use this if the binary is for a platform whose ELF**
  loader only looks at section headers, but whose toolchain generates program headers anyway.

is_default = True

__init__(**args*, *addend=None*, *debug_symbols=None*, *discard_section_headers=False*, *discard_program_headers=False*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

addr_to_line: SortedDict[int, set[tuple[int, int]]]

variables: list[*Variable*] | None

compilation_units: list[*CompilationUnit*] | None

functions_debug_info: dict[int, *Subprogram*]

extern_size_hints: dict[str, int]

close()

classmethod check_compatibility(*spec*, *obj*)

Performs a minimal static load of `spec` and returns whether it's compatible with other_obj

classmethod check_magic_compatibility(*stream*)

Check if a stream of bytes contains the same magic number as the main object

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

static extract_arch(*reader*)

property initializers

Stub function. Should be overridden by backends that can provide initializer functions that ought to be run before execution reaches the entry point. Addresses should be rebased.

property finalizers

Stub function. Like initializers, but with finalizers.

property symbols_by_name

get_symbol(*symid*, *symbol_table=None*)

Gets a Symbol object for the specified symbol.

Parameters

symid – Either an index into .dynsym or the name of a symbol.

rebase(*new_base*)

Rebase backend's regions to the new base where they were mapped by the loader

class cle.backends.elf.**ELFCore**

Bases: *ELF*

Loader class for ELF core files.

One key pain point when analyzing a core dump generated on a remote machine is that the paths to binaries are absolute (and may not exist or be the same on your local machine).

Therefore, you can use the options `remote_file_mapping to specify a `dict` mapping (easy if there are a small number of mappings) or `remote_file_mapper` to specify a function that accepts a remote file name and returns the local file name (useful if there are many mappings).

If you specify both `remote_file_mapping` and `remote_file_mapper`, `remote_file_mapping` is applied first, then the result is passed to `remote_file_mapper`.

Parameters

- **executable** – Optional path to the main binary of the core dump. If not supplied, ELFCore will attempt to figure it out automatically from the core dump.
- **remote_file_mapping** – Optional dict that maps specific file names in the core dump to other file names.
- **remote_file_mapper** – Optional function that is used to map every file name in the core dump to whatever is returned from this function.

is_default = True

__init__(**args*, *executable=None*, *remote_file_mapping=None*, *remote_file_mapper=None*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load

- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

property threads

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This property should contain a list of names for these threads, which should be unique.

thread_registers(*thread=None*)

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This method should return the register file for a given thread (as named in `Backend.threads`) as a dict mapping register names (as seen in archinfo) to numbers. If the thread is not specified, it should return the context for a "default" thread. If there are no threads, it should return an empty dict.

class cle.backends.elf.**MetaELF**

Bases: *Backend*

A base class that implements functions used by all backends that can load an ELF.

__init__(**args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

supported_filetypes = ['elf']

property plt

Maps names to addresses.

property reverse_plt

Maps addresses to names.

property is_ppc64_abiv1

Returns whether the arch is PowerPC64 ABIv1.

Returns
True if PowerPC64 ABIv1, False otherwise.

property is_ppc64_abiv2

Returns whether the arch is PowerPC64 ABIv2.

Returns
True if PowerPC64 ABIv2, False otherwise.

property ppc64_initial_rtoc

Get initial rtoc value for PowerPC64 architecture.

static extract_soname(*path*)

Extracts the shared object identifier from the path, or returns None if it cannot.

class cle.backends.elf.metaelf.**Relro**

Bases: Enum

NONE = 0

PARTIAL = 1

FULL = 2

class cle.backends.elf.symbol.**ELFSymbol**

Bases: *Symbol*

Represents a symbol for the ELF format.

Variables

- **binding** (*str*) – The binding of this symbol as an ELF enum string
- **section** – The section associated with this symbol, or None
- **_subtype** – The ELFSymbolType of this symbol

__init__(*owner*, *symb*)

Not documenting this since if you try calling it, you're wrong.

property subtype: *ELFSymbolType*

A subclass' ABI-specific types

class cle.backends.elf.symbol_type.**ELFSymbolType**

Bases: *SymbolSubType*

ELF-specific symbol types

STT_NOTYPE = (0, None)

STT_OBJECT = (1, None)

STT_FUNC = (2, None)

STT_SECTION = (3, None)

STT_FILE = (4, None)

STT_COMMON = (5, None)

STT_TLS = (6, None)

STT_LOOS = (10, None)

STT_HIOS = (12, None)

STT_LOPROC = (13, None)

STT_HIPROC = (15, None)

STT_GNU_IFUNC = (10, 'gnu')

__init__(**args*)

property elf_value

property os_proc

property is_custom_os_proc

to_base_type()

A subclass' ABI-specific mapping to :SymbolType:

__new__(*value*)

This is just a nice way to allow for just specifying the *int* for default types: *ELFSymbolType(10)* rather than *ELFSymbolType((10,None))*.

Idea courtesy: https://stackoverflow.com/q/24105268/1137728.

We don't need to implement the *str* parsing like the SO link above since *Enum* already has built-in item access: *ELFSymbolType['STT_FUNC']*.

class cle.backends.elf.regions.**ELFSegment**

Bases: *Segment*

Represents a segment for the ELF format.

__init__(*readelf_seg*, *relro=False*)

property is_readable

property is_writable

property is_executable

property is_relro

class cle.backends.elf.regions.**ELFSection**

Bases: *Section*

SHF_WRITE = 1

SHF_ALLOC = 2

SHF_EXECINSTR = 4

SHF_STRINGS = 32

SHT_NULL = 'SHT_NULL'

__init__(*readelf_sec*, *remap_offset=0*)

Parameters

- **name** (*str*) – The name of the section
- **offset** (*int*) – The offset into the binary file this section begins
- **vaddr** (*int*) – The address in virtual memory this section begins
- **size** (*int*) – How large this section is

property is_readable

Whether this section has read permissions

property is_active

property is_writable

Whether this section has write permissions

property occupies_memory

property is_executable

Whether this section has execute permissions

property is_strings

property only_contains_uninitialized_data

Whether this section is initialized to zero after the executable is loaded.

class cle.backends.elf.variable.**Variable**

Bases: object

Variable for DWARF from a DW_TAG_variable or DW_TAG_formal_parameter

Variables

- **name** (*str*) – The name of the variable
- **relative_addr** (int | None) – The relative addr (base addr depends on the type)
- **lexical_block** (*LexicalBlock* | None) – For a local variable, the lexical block where the variable is declared

__init__(*elf_object:* ELF)

Parameters
elf_object (ELF)

static from_die(*die: DIE*, *expr_parser*, *elf_object:* ELF, *lexical_block:* LexicalBlock | *None* = *None*, *namespace: list[str]* | *None* = *None*)

Parameters

- **die** (*DIE*)
- **elf_object** (ELF)
- **lexical_block** (LexicalBlock | *None*)
- **namespace** (*list[str]* | *None*)

rebased_addr_from_cfa(*cfa: int*) → int

The address of this variable in the global memory.

Parameters
cfa (int) – The canonical frame address as described by the DWARF standard.

Return type
int

property rebased_addr

property addr

Please use 'relative_addr' or 'rebased_addr' instead.

property type: *VariableType*

property sort: str

class cle.backends.elf.variable.**MemoryVariable**

Bases: *Variable*

This includes all variables that are not on the stack and not in a register. So all global variables, and also local static variables in C!

__init__(*elf_object:* ELF, *relative_addr*)

Parameters
elf_object (ELF)

property rebased_addr

property sort: str

class cle.backends.elf.variable.**StackVariable**

Bases: *Variable*

Stack Variable from DWARF.

__init__(*elf_object:* ELF, *relative_addr*)

Parameters
elf_object (ELF)

rebased_addr_from_cfa(*cfa: int*)

The address of this variable in the global memory.

Parameters
cfa (int) – The canonical frame address as described by the DWARF standard.

property sort: str

class cle.backends.elf.variable.**RegisterVariable**

Bases: *Variable*

Register Variable from DWARF.

__init__(*elf_object:* ELF, *register_addr*)

Parameters
elf_object (ELF)

property sort: str

cle.backends.elf.variable_type.**resolve_reference_addr**(*die: DIE*, *attr_name: str*) → int

Resolves a reference attribute to the underlying DIE location :type die: DIE :param die: The DIE containing the reference attribute :type attr_name: str :param attr_name: The name of the attribute as a string

Return type
int

Returns
The address of the DIE referred to by the reference

Parameters

- **die** (*DIE*)
- **attr_name** (*str*)

class cle.backends.elf.variable_type.**VariableType**

Bases: object

Entry class for DW_TAG_xxx_type

Parameters

- **name** (str) – name of the type
- **byte_size** (int) – amount of bytes the type take in memory
- **elf_object** – elf object to reference to (useful for pointer,...)

Variables

- **name** – name of the type
- **byte_size** – amount of bytes the type take in memory

__init__(*name: str*, *byte_size: int*, *elf_object*)

Parameters

- **name** (*str*)
- **byte_size** (*int*)

static read_from_die(*die: DIE*, *elf_object*)

entry method to read a DW_TAG_xxx_type

Parameters
die (*DIE*)

static supported_die(*die: DIE*) → bool

Return type
bool

Parameters
die (*DIE*)

class cle.backends.elf.variable_type.**PointerType**

Bases: *VariableType*

Entry class for DW_TAG_pointer_type. It is inherited from VariableType

Parameters

- **byte_size** (int) – amount of bytes the type take in memory
- **elf_object** – elf object to reference to (useful for pointer,...)
- **referenced_offset** (int) – type of the referenced as offset in the compilation_unit

__init__(*byte_size: int*, *elf_object*, *referenced_offset: int*)

Parameters

- **byte_size** (*int*)
- **referenced_offset** (*int*)

classmethod read_from_die(*die: DIE*, *elf_object*)

read an entry of DW_TAG_pointer_type. return None when there is no byte_size or type attribute.

Parameters
die (*DIE*)

property referenced_type

attribute to get the referenced type. Return None if the type is not loaded

class cle.backends.elf.variable_type.**BaseType**

Bases: *VariableType*

Entry class for DW_TAG_base_type. It is inherited from VariableType

classmethod read_from_die(*die: DIE*, *elf_object*)

read an entry of DW_TAG_base_type. return None when there is no byte_size attribute.

Parameters
die (*DIE*)

class cle.backends.elf.variable_type.**StructType**

Bases: *VariableType*

Entry class for DW_TAG_structure_type. It is inherited from VariableType

Parameters

- **name** (str) – name of the type
- **byte_size** (int) – amount of bytes the type take in memory
- **elf_object** – elf object to reference to (useful for pointer,...)

__init__(*name: str*, *byte_size: int*, *elf_object*, *members*)

Parameters

- **name** (*str*)
- **byte_size** (*int*)

classmethod read_from_die(*die: DIE*, *elf_object*)

read an entry of DW_TAG_structure_type. return None when there is no byte_size attribute.

Parameters
die (*DIE*)

class cle.backends.elf.variable_type.**UnionType**

Bases: *StructType*

Entry class for DW_TAG_union_type. Inherits from StructType to make it trivial.

class cle.backends.elf.variable_type.**StructMember**

Bases: object

Entry class for DW_TAG_member. This is not a type but a named member inside a struct. Use the property *type* to get its variable type.

Parameters

- **name** (str) – name of the member
- **addr_offset** (int) – address offset of the member in the struct
- **elf_object** – elf object to reference to (useful for pointer,...)
- **type_offset** – type as offset in the compilation_unit

Variables
name – name of the member

__init__(*name: str*, *addr_offset: int*, *type_offset*, *elf_object*)

Parameters

- **name** (*str*)
- **addr_offset** (*int*)

classmethod **read_from_die**(*die: DIE*, *elf_object*)

read an entry of DW_TAG_member_type. return None when there is no type attribute.

Parameters
die (*DIE*)

property **type**

attribute to get the type of the member. Return None if the type is not loaded

class cle.backends.elf.variable_type.**ArrayType**

Bases: *VariableType*

Entry class for DW_TAG_array_type. It is inherited from VariableType

Parameters

- **byte_size** – amount of bytes the type take in memory
- **elf_object** – elf object to reference to (useful for pointer,...)
- **element_offset** – type of the array elements as offset in the compilation_unit

__init__(*byte_size*, *elf_object*, *element_offset*)

classmethod **read_from_die**(*die: DIE*, *elf_object*)

read an entry of DW_TAG_array_type. return None when there is no type attribute.

Parameters
die (*DIE*)

property **element_type**

class cle.backends.elf.variable_type.**TypedefType**

Bases: *VariableType*

Entry class for DW_TAG_typedef. Inherits from VariableType.

Parameters

- **name** (*str*) – name of the new type
- **elf_object** – elf object to reference to (useful for pointer,...)
- **type_offset** – type as offset in the compilation_unit

__init__(*name: str*, *byte_size*, *elf_object*, *type_offset*)

Parameters
name (*str*)

classmethod **read_from_die**(*die: DIE*, *elf_object*)

read an entry of DW_TAG_member_type. return None when there is no type attribute.

Parameters
die (*DIE*)

property type

attribute to get the type of the member. Return None if the type is not loaded

References

- http://www.hexblog.com/wp-content/uploads/2012/06/Recon-2012-Skochinsky-Compiler-Internals.pdf
- https://www.airs.com/blog/archives/460
- https://www.airs.com/blog/archives/464

class cle.backends.elf.lsda.**ExceptionTableHeader**

Bases: object

__init__(*lp_start*, *ttype_encoding*, *ttype_offset*, *call_site_encoding*, *call_site_table_len*)

lp_start

ttype_encoding

ttype_offset

call_site_encoding

call_site_table_len

class cle.backends.elf.lsda.**CallSiteEntry**

Bases: object

__init__(*cs_start*, *cs_len*, *cs_lp*, *cs_action*)

cs_start

cs_len

cs_lp

cs_action

class cle.backends.elf.lsda.**LSDAExceptionTable**

Bases: object

LSDA exception table parser.

TODO: Much of this class should be eventually moved to pyelftools.

__init__(*stream*, *bits*, *little_endian=True*)

parse_lsda(*address*, *offset*)

class cle.backends.elf.hashtable.**ELFHashTable**

Bases: object

Functions to do lookup from a HASH section of an ELF file.

Information: http://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-48031.html

__init__(*symtab*, *stream*, *offset*, *arch*)

Parameters

- **symtab** – The symbol table to perform lookups from (as a pyelftools SymbolTableSection).

- **stream** – A file-like object to read from the ELF's memory.
- **offset** – The offset in the object where the table starts.
- **arch** – The ArchInfo object for the ELF file.

get(*k*)

Perform a lookup. Returns a pyelftools Symbol object, or None if there is no match.

Parameters
k – The string to look up.

static elf_hash(*key*)

class cle.backends.elf.hashtable.**GNUHashTable**

Bases: object

Functions to do lookup from a GNU_HASH section of an ELF file.

Information: https://blogs.oracle.com/ali/entry/gnu_hash_elf_sections

__init__(*symtab*, *stream*, *offset*, *arch*)

Parameters

- **symtab** – The symbol table to perform lookups from (as a pyelftools SymbolTableSection).
- **stream** – A file-like object to read from the ELF's memory.
- **offset** – The offset in the object where the table starts.
- **arch** – The ArchInfo object for the ELF file.

get(*k*)

Perform a lookup. Returns a pyelftools Symbol object, or None if there is no match.

Parameters
k – The string to look up

static gnu_hash(*key*)

class cle.backends.elf.subprogram.**LexicalBlock**

Bases: object

A lexical block is a sequence of source statements, e.g. a while/for loop or an if statement or some bracketed block.

Corresponds to a DW_TAG_LexicalBlock in DWARF.

Parameters

- **super_block** – The lexical block which contains this block
- **low_pc** (int | None) – The relative start address of the block
- **high_pc** (int | None) – The relative end address of the block

Variables

- **low_pc** – The relative start address of the subprogram
- **high_pc** – The relative end address of the subprogram
- **child_blocks** (list[*LexicalBlock*]) – Lexical blocks inside this block (only direct childs)

__init__(*low_pc: int | None*, *high_pc: int | None*, *ranges: list[tuple[int, int]] | None = None*, *source_file: str | None = None*, *source_line: int | None = None*) → None

Parameters

- **low_pc** (*int | None*)
- **high_pc** (*int | None*)
- **ranges** (*list[tuple[int, int]] | None*)
- **source_file** (*str | None*)
- **source_line** (*int | None*)

Return type
None

rebase(*delta: int*)

Parameters
delta (*int*)

class cle.backends.elf.subprogram.**Subprogram**

Bases: *LexicalBlock*

DW_TAG_subprogram for DWARF. The behavior is mostly inherited from LexicalBlock to avoid redundancy.

Parameters

- **name** (str | None) – The name of the function/program
- **low_pc** (int | None) – The relative start address of the subprogram
- **high_pc** (int | None) – The relative end address of the subprogram

Variables

- **name** – The name of the function/program
- **local_variables** (list[*Variable*]) – All local variables in a Subprogram (they may reside in serveral child blocks)

__init__(*name: str | None*, *low_pc: int | None*, *high_pc: int | None*, *ranges: list[tuple[int, int]] | None = None*, *source_file: str | None = None*, *source_line: int | None = None*) → None

Parameters

- **name** (*str | None*)
- **low_pc** (*int | None*)
- **high_pc** (*int | None*)
- **ranges** (*list[tuple[int, int]] | None*)
- **source_file** (*str | None*)
- **source_line** (*int | None*)

Return type
None

rebase(*delta: int*)

Parameters
delta (*int*)

class cle.backends.elf.compilation_unit.**CompilationUnit**

Bases: object

CompilationUnit for DWARF See http://dwarfstd.org/doc/DWARF5.pdf page 60

__init__(*name*, *comp_dir*, *low_pc*, *high_pc*, *language*, *elf_object*, *ranges*)

property min_addr

property max_addr

2.3.2 PE

class cle.backends.**PE**

Bases: *Backend*

Representation of a PE (i.e. Windows) binary.

Useful backend options:

- `debug_symbols`: Provides the path to a PDB file which contains the binary's debug symbols
- `debug_symbol_dirs`: List of directories to search for PDB files (searched before symbol servers)
- **debug_symbol_path_str: A string indicating symbol search paths, which may be provided in the** _NT_SYMBOL_PATH format.
- **download_debug_symbols: Whether to attempt downloading debug symbols from symbol servers (if provided) or** not. Default to False.
- **download_debug_symbol_confirm: A callable that takes a URL string and returns True if downloading the debug** symbol from the URL is allowed by the user, False otherwise.
- **download_debug_symbol_progress: A callable that takes two integer arguments: bytes downloaded and total bytes.** This callable is called periodically to report download progress.
- **search_microsoft_symserver: Whether to include the Microsoft symbol server in symbol searches. Default to** True. Requires `download_debug_symbols` to be True to have any effect.

is_default = True

__init__(**args*, *debug_symbols=None*, *debug_symbol_dirs=None*, *debug_symbol_path_str: str | None = None*, *download_debug_symbols: bool = False*, *download_debug_symbol_confirm: Callable[[str], bool] | None = None*, *download_debug_symbol_progress: Callable[[int, int | None], bool] | None = None*, *search_microsoft_symserver: bool = True*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable
- **debug_symbol_path_str** (*str | None*)
- **download_debug_symbols** (*bool*)
- **download_debug_symbol_confirm** (*Callable[[str], bool] | None*)

- **download_debug_symbol_progress** (*Callable[[int, int | None], bool] | None*)
- **search_microsoft_symserver** (*bool*)

classmethod is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

classmethod check_magic_compatibility(*stream*)

Check if a stream of bytes contains the same magic number as the main object

classmethod check_compatibility(*spec*, *obj*)

Performs a minimal static load of `spec` and returns whether it's compatible with other_obj

close()

get_symbol(*name*)

Look up the symbol with the given name. Symbols can be looked up by ordinal with the name `"ordinal. %d" % num`

load_symbols_from_pdb(*pdb_path*)

Load available symbols from PDB at *pdb_path*

class cle.backends.pe.regions.**PESection**

Bases: *Section*

Represents a section for the PE format.

__init__(*pe_section*, *remap_offset=0*, *name: str | None = None*)

Parameters

- **name** (`str | None`) – The name of the section
- **offset** (*int*) – The offset into the binary file this section begins
- **vaddr** (*int*) – The address in virtual memory this section begins
- **size** (*int*) – How large this section is

property is_readable

Whether this section has read permissions

property is_writable

Whether this section has write permissions

property is_executable

Whether this section has execute permissions

property only_contains_uninitialized_data

Whether this section is initialized to zero after the executable is loaded.

class cle.backends.pe.symbol.**WinSymbol**

Bases: *Symbol*

Represents a symbol for the PE format.

__init__(*owner*, *name*, *addr*, *is_import*, *is_export*, *ordinal_number*, *forwarder*, *symbol_type=SymbolType.TYPE_FUNCTION*)

Not documenting this since if you try calling it, you're wrong.

resolve_forwarder()

If this symbol is a forwarding export, return the symbol the forwarding refers to, or None if it cannot be found

2.3.3 Mach-O

class cle.backends.macho.macho.**MachO**

Bases: *Backend*

Mach-O binaries for CLE

The Mach-O format is notably different from other formats. Specifically:

- Sections are always part of a segment, so *self.sections* will be empty.
- Symbols cannot be categorized like in ELF.
- Symbol resolution must be handled by the binary.
- Rebasing in dyld is implemented by adding a small slide to addresses inside the binary, instead of changing the base address of the binary. Consequently, the addresses are absolute rather than relative. CLE requires relative addresses, leading to numerous *AT.from_lva().to_rva()* calls in this backend.

is_default = True

MH_MAGIC_64 = 4277009103

MH_CIGAM_64 = 3489328638

MH_MAGIC = 4277009102

MH_CIGAM = 3472551422

__init__(**args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

ncmds: int

sizeofcmds: int

property stubs

Maps names to addresses.

property plt

Maps names to addresses.

property reverse_stubs

Maps addresses to names.

property reverse_plt

Maps addresses to names.

set_arch(*arch*)

property min_addr

This returns the lowest virtual address contained in any loaded segment of the binary.

classmethod check_compatibility(*spec*, *obj*)

Performs a minimal static load of `spec` and returns whether it's compatible with other_obj

static extract_soname(*path*)

Extracts the shared object identifier from the path, or returns None if it cannot.

classmethod is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

is_thumb_interworking(*address*)

Returns true if the given address is a THUMB interworking address

decode_thumb_interworking(*address*)

Decodes a thumb interworking address

find_segment_by_name(*name*)

do_binding()

get_string(*start*)

Loads a string from the string table

parse_lc_str(*f*, *start*, *limit: int | None = None*)

Parses a lc_str data structure

Parameters
limit (*int | None*)

S = ~S

get_symbol_by_address_fuzzy(*address*)

Locates a symbol by checking the given address against sym.addr, sym.bind_xrefs and sym.symbol_stubs

get_symbol(*name*, *include_stab=False*, *fuzzy=False*)

Returns all symbols matching name.

Note that especially when include_stab=True there may be multiple symbols with the same name, therefore this method always returns an array.

Parameters

- **name** – the name of the symbol
- **include_stab** – Include debugging symbols NOT RECOMMENDED
- **fuzzy** – Replace exact match with "contains"-style match

get_symbol_by_insertion_order(*idx: int*) → *AbstractMachOSymbol*

Parameters
idx (*int*) – idx when this symbol was inserted

Return type
AbstractMachOSymbol

Returns

get_segment_by_name(*name*)

Searches for a MachOSegment with the given name and returns it :type name: :param name: Name of the sought segment :return: MachOSegment or None

class cle.backends.macho.macho.**SymbolList**

Bases: SortedKeyList

Special data structure that extends SortedKeyList to allow looking up a MachO library by name and ordinal quickly without having to iterate over the whole list

__init__(***kwargs*)

Initialize sorted-key list instance.

Optional *iterable* argument provides an initial iterable of values to initialize the sorted-key list.

Optional *key* argument defines a callable that, like the *key* argument to Python's *sorted* function, extracts a comparison key from each value. The default is the identity function.

Runtime complexity: *O(n*log(n))*

```
>>> from operator import neg
>>> skl = SortedKeyList(key=neg)
>>> skl
SortedKeyList([], key=<built-in function neg>)
>>> skl = SortedKeyList([3, 1, 2], key=neg)
>>> skl
SortedKeyList([3, 2, 1], key=<built-in function neg>)
```

Parameters

- **iterable** – initial values (optional)
- **key** – function used to extract comparison key (optional)

add(*value:* AbstractMachOSymbol)

Add *value* to sorted-key list.

Runtime complexity: *O(log(n))* – approximate.

```
>>> from operator import neg
>>> skl = SortedKeyList(key=neg)
>>> skl.add(3)
>>> skl.add(1)
>>> skl.add(2)
>>> skl
SortedKeyList([3, 2, 1], key=<built-in function neg>)
```

Parameters

value (*AbstractMachOSymbol*) – value to add to sorted-key list

get_by_name_and_ordinal(*name: str*, *ordinal: int*, *include_stab=False*) → list[*AbstractMachOSymbol*]

Return type

list[*AbstractMachOSymbol*]

Parameters

- **name** (*str*)

- **ordinal** (*int*)

class cle.backends.macho.symbol.**SymbolType**

Bases: Enum

ABI-agnostic symbol types

TYPE_OTHER = 0

TYPE_NONE = 1

TYPE_FUNCTION = 2

TYPE_OBJECT = 3

TYPE_SECTION = 4

TYPE_TLS_OBJECT = 5

TYPE_FUNCTION_OR_OBJECT = 6

class cle.backends.macho.symbol.**AbstractMachOSymbol**

Bases: *Symbol*

Base class for Mach-O symbols. Defines the minimum common properties all types of mach-o symbols must have

owner: *MachO*

__init__(*owner:* Backend, *name: str*, *relative_addr: int*, *size: int*, *sym_type:* SymbolType)

Not documenting this since if you try calling it, you're wrong.

Parameters

- **owner** (Backend)
- **name** (*str*)
- **relative_addr** (*int*)
- **size** (*int*)
- **sym_type** (SymbolType)

property library_ordinal

property is_stab: bool

property library_name: str | None

property library_base_name: str | None

class cle.backends.macho.symbol.**SymbolTableSymbol**

Bases: *AbstractMachOSymbol*

"Regular" symbol. Made to be (somewhat) compatible with backends.Symbol. A SymbolTableSymbol is an entry in the binary's symbol table.

Note that ELF-specific fields from backends.Symbol are not used and semantics of the remaining fields differ in many cases. As a result most stock functionality from Angr and related libraries WILL NOT WORK PROPERLY on MachOSymbol.

Much of the code below is based on heuristics as official documentation is sparse, consider yourself warned!

The relevant struct with documentation is nlist_64 defined in mach-o/nlist.h

__init__(*owner: MachO*, *symtab_offset*, *n_strx*, *n_type*, *n_sect*, *n_desc*, *n_value*)

Not documenting this since if you try calling it, you're wrong.

Parameters
owner (Mach0)

property library_name: str | None

property segment_name

property section_name

property value

property referenced_symbol_index

For indirect symbols n_value contains an index into the string table indicating the referenced symbol's name

property is_weak

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property is_function

Whether this symbol is a function

property is_stab: bool

property is_private_external: bool

property is_external: bool

property sym_type

property is_common

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property common_align

property reference_type

property library_ordinal

property is_no_dead_strip

property is_desc_discarded

property is_weak_referenced

property is_weak_defined

property is_reference_to_weak

property is_thumb_definition

property is_symbol_resolver

property is_alt_entry

class cle.backends.macho.symbol.**BindingSymbol**

Bases: *AbstractMachOSymbol*

"Binding" symbol. Made to be (somewhat) compatible with backends.Symbol. A BindingSymbol is an imported symbol discovered during the binding process.

Note that ELF-specific fields from backends.Symbol are not used and semantics of the remaining fields differ in many cases. As a result most stock functionality from Angr and related libraries WILL NOT WORK PROPERLY on MachOSymbol.

Much of the code below is based on heuristics as official documentation is sparse, consider yourself warned!

__init__(*owner*, *name*, *lib_ordinal*)

Not documenting this since if you try calling it, you're wrong.

property library_name: str | None

property is_function

Whether this symbol is a function

demangled_name()

property library_ordinal

class cle.backends.macho.symbol.**DyldBoundSymbol**

Bases: *AbstractMachOSymbol*

The new kind of symbol handling introduced with ios15

owner: *MachO*

__init__(*owner*, *name*, *lib_ordinal*)

Based on the constructor of BindingSymbol

property library_name: str | None

property is_function

Whether this symbol is a function

demangled_name()

property library_ordinal

class cle.backends.macho.section.**MachOSection**

Bases: *Section*

Mach-O Section, only defined within the context of a Mach-O Segment.

- offset is the offset into the file the region starts
- vaddr (or just addr) is the virtual address
- filesize (or just size) is the size of the region in the file
- memsize (or vsize) is the size of the region when loaded into memory
- segname is the corresponding segment's name without padding
- sectname is the section's name without padding
- align is the sections alignment as a power of 2

- reloff is the file offset to the section's relocation entries
- nreloc is the number of relocation entries for this section
- flags is a bit vector containing per-section flags
- r1 and r2 are values for the reserved1 and reserved2 fields respectively

__init__(*offset*, *vaddr*, *size*, *vsize*, *segname*, *sectname*, *align*, *reloff*, *nreloc*, *flags*, *r1*, *r2*, *parent_segment:* MachOSegment | *None* = *None*)

Parameters

- **name** (*str*) – The name of the section
- **offset** (*int*) – The offset into the binary file this section begins
- **vaddr** (*int*) – The address in virtual memory this section begins
- **size** (*int*) – How large this section is
- **parent_segment** (MachOSegment | *None*)

property full_name: str

property type

property attributes

property is_readable

Always true, because sections should always be readable :return:

property is_writable

Returns the permission of the parent segment, because MachO sections simply inherit that :return:

property is_executable

Returns the permission of the parent segment, because MachO sections simply inherit that :return:

property only_contains_uninitialized_data

I actually don't know if this is true, but it seems like a saner assumption than true :return:

class cle.backends.macho.segment.**MachOSegment**

Bases: *Segment*

Mach-O Segment

- offset is the offset into the file the region starts
- vaddr (or just addr) is the virtual address
- filesize (or just size) is the size of the region in the file
- memsize (or vsize) is the size of the region when loaded into memory
- segname is the segment's name without padding
- nsect is the number of sections contained in this segment
- sections is an array of MachOSections
- flags is a bit vector containing per-segment flags
- initprot and maxprot are initial and maximum permissions respectively

__init__(*offset*, *vaddr*, *size*, *vsize*, *segname*, *nsect*, *sections*, *flags*, *initprot*, *maxprot*)

get_section_by_name(*name*)

Searches for a section by name within this segment :type name: :param name: Name of the section :return: MachOSection or None

property is_readable

property is_writable

property is_executable

cle.backends.macho.binding.**chh**(*x*)

cle.backends.macho.binding.**read_uleb**(*blob: bytes*, *offset: int*) → tuple[int, int]

Reads a number encoded as uleb128

Return type
tuple[int, int]

Parameters

- **blob** (*bytes*)
- **offset** (*int*)

cle.backends.macho.binding.**read_sleb**(*blob*, *offset*)

Reads a number encoded as sleb128

class cle.backends.macho.binding.**BindingState**

Bases: object

State object

__init__(*is_64*)

add_address_ov(*address*, *addend*)

this is a very ugly klugde. It is needed because dyld relies on overflow semantics and represents several negative offsets through BIG ulebs

check_address_bounds()

class cle.backends.macho.binding.**BindingHelper**

Bases: object

Factors out binding logic from MachO. Intended to work in close conjunction with MachO not for standalone use

__init__(*binary*)

binary: *MachO*

do_normal_bind(*blob: bytes*)

Performs non-lazy, non-weak bindings :type blob: bytes :param blob: Blob containing binding opcodes

Parameters
blob (*bytes*)

do_lazy_bind(*blob*)

Performs lazy binding

do_rebases(*blob: bytes*)

Handles the rebase blob Implementation based closely on ImageLoaderMachOCompressed::rebase from dyld https://github.com/apple-opensource/dyld/blob/e3f88907bebb8421f50f0943595f6874de70ebe0/src/ImageLoaderMach
L463

Parameters
blob (bytes)

Returns

static read_uleb(*blob*, *offset*) → tuple[int, int]

little helper to read ulebs, that also returns the new index :type blob: :param blob: :type offset: :param offset:

Return type
tuple[int, int]

Returns

rebase_at(*address: int*, *ty: RebaseType*)

Parameters

- **address** (*int*)
- **ty** (*RebaseType*)

cle.backends.macho.binding.**n_opcode_done**(*s:* BindingState, *_b:* MachO, *_i: int*, *_blob: bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **_b** (MachO)
- **_i** (*int*)
- **_blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_set_dylib_ordinal_imm**(*s:* BindingState, *_b:* MachO, *i: int*, *_blob: bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **_b** (MachO)
- **i** (*int*)
- **_blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_set_dylib_ordinal_uleb**(*s:* BindingState, *_b:* MachO, *_i: int*, *blob: bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **_b** (MachO)
- **_i** (*int*)
- **blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_set_dylib_special_imm**(*s:* BindingState, *_b:* MachO, *i:* *int*, *_blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **_b** (MachO)
- **i** (*int*)
- **_blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_set_trailing_flags_imm**(*s:* BindingState, *_b:* MachO, *i:* *int*, *blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **_b** (MachO)
- **i** (*int*)
- **blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_set_type_imm**(*s:* BindingState, *_b:* MachO, *i:* *int*, *_blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **_b** (MachO)
- **i** (*int*)
- **_blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_set_addend_sleb**(*s:* BindingState, *_b:* MachO, *_i:* *int*, *blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)

- **_b** (MachO)
- **_i** (*int*)
- **blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_set_segment_and_offset_uleb**(*s:* BindingState, *b:* MachO, *i:* *int*, *blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **b** (MachO)
- **i** (*int*)
- **blob** (*bytes*)

cle.backends.macho.binding.**l_opcode_set_segment_and_offset_uleb**(*s:* BindingState, *b:* MachO, *i:* *int*, *blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **b** (MachO)
- **i** (*int*)
- **blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_add_addr_uleb**(*s:* BindingState, *_b:* MachO, *_i:* *int*, *blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **_b** (MachO)
- **_i** (*int*)
- **blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_do_bind**(*s:* BindingState, *b:* MachO, *_i:* *int*, *_blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **b** (MachO)
- **_i** (*int*)

- **_blob** (*bytes*)

cle.backends.macho.binding.**l_opcode_do_bind**(*s:* BindingState, *b:* MachO, *_i:* *int*, *_blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **b** (MachO)
- **_i** (*int*)
- **_blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_do_bind_add_addr_uleb**(*s:* BindingState, *b:* MachO, *_i:* *int*, *blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **b** (MachO)
- **_i** (*int*)
- **blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_do_bind_add_addr_imm_scaled**(*s:* BindingState, *b:* MachO, *i:* *int*, *_blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **b** (MachO)
- **i** (*int*)
- **_blob** (*bytes*)

cle.backends.macho.binding.**n_opcode_do_bind_uleb_times_skipping_uleb**(*s:* BindingState, *b:* MachO, *_i:* *int*, *blob:* *bytes*) → *BindingState*

Return type
BindingState

Parameters

- **s** (BindingState)
- **b** (MachO)
- **_i** (*int*)
- **blob** (*bytes*)

class cle.backends.macho.binding.**MachOSymbolRelocation**

Bases: *Relocation*

Generic Relocation for MachO. It handles relocations that point to symbols

symbol: *AbstractMachOSymbol*

__init__(*owner:* MachO, *symbol:* AbstractMachOSymbol, *relative_addr:* *int*, *data*)

Parameters

- **owner** (MachO)
- **symbol** (AbstractMachOSymbol)
- **relative_addr** (*int*)

resolve_symbol(*solist*, *thumb=False*, *extern_object=None*, ***kwargs*)

property dest_addr

property value

class cle.backends.macho.binding.**MachOPointerRelocation**

Bases: *Relocation*

A relocation for a pointer without any associated symbol These are either generated while handling the rebase blob, or while parsing chained fixups

__init__(*owner:* MachO, *relative_addr:* *int*, *data*)

Parameters

- **owner** (*MachO*)
- **relative_addr** (int) – the relative addr where this relocation is located
- **data** – the rebase offset relative to the linked base

property value

resolve_symbol(*solist*, *thumb=False*, *extern_object=None*, ***kwargs*)

This relocation has no associated symbol, so we don't need to resolve it. :type solist: :param solist: :type thumb: :param thumb: :type extern_object: :param extern_object: :type kwargs: :param kwargs: :return:

cle.backends.macho.binding.**default_binding_handler**(*state:* BindingState, *binary:* MachO)

Binds location to the symbol with the given name and library ordinal

Parameters

- **state** (BindingState)
- **binary** (Mach0)

class cle.backends.macho.structs.**HelperStruct**

Bases: Structure

Subclass of ctypes.Structure that adds a helpful repr method for debugging

class cle.backends.macho.structs.**DyldImportFormats**

Bases: IntEnum

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L249-L254

DYLD_CHAINED_IMPORT = 1

DYLD_CHAINED_IMPORT_ADDEND = 2

DYLD_CHAINED_IMPORT_ADDEND64 = 3

__new__(*value*)

class cle.backends.macho.structs.**DyldChainedPtrFormats**

Bases: IntEnum

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L89-L104

DYLD_CHAINED_PTR_ARM64E = 1

DYLD_CHAINED_PTR_64 = 2

DYLD_CHAINED_PTR_32 = 3

DYLD_CHAINED_PTR_32_CACHE = 4

DYLD_CHAINED_PTR_32_FIRMWARE = 5

DYLD_CHAINED_PTR_64_OFFSET = 6

DYLD_CHAINED_PTR_ARM64E_KERNEL = 7

DYLD_CHAINED_PTR_64_KERNEL_CACHE = 8

DYLD_CHAINED_PTR_ARM64E_USERLAND = 9

DYLD_CHAINED_PTR_ARM64E_FIRMWARE = 10

DYLD_CHAINED_PTR_X86_64_KERNEL_CACHE = 11

DYLD_CHAINED_PTR_ARM64E_USERLAND24 = 12

__new__(*value*)

class cle.backends.macho.structs.**dyld_chained_ptr_arm64e_auth_rebase**

Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L128-L138

addrDiv
: Structure/Union member

auth
: Structure/Union member

bind
: Structure/Union member

diversity
: Structure/Union member

key
: Structure/Union member

next
: Structure/Union member

target

Structure/Union member

class cle.backends.macho.structs.**dyld_chained_ptr_arm64e_auth_bind**

Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L140-L151

addrDiv

Structure/Union member

auth

Structure/Union member

bind

Structure/Union member

diversity

Structure/Union member

key

Structure/Union member

next

Structure/Union member

ordinal

Structure/Union member

zero

Structure/Union member

class cle.backends.macho.structs.**dyld_chained_ptr_arm64e_rebase**

Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L107-L115

auth

Structure/Union member

bind

Structure/Union member

high8

Structure/Union member

next

Structure/Union member

target

Structure/Union member

class cle.backends.macho.structs.**dyld_chained_ptr_arm64e_bind**

Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L117-L126

addend

Structure/Union member

auth
Structure/Union member

bind
Structure/Union member

next
Structure/Union member

ordinal
Structure/Union member

zero
Structure/Union member

class cle.backends.macho.structs.**dyld_chained_ptr_arm64e_bind24**
Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L164-L173

class cle.backends.macho.structs.**dyld_chained_ptr_arm64e_auth_bind24**
Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L175-L186

addrDiv
Structure/Union member

auth
Structure/Union member

bind
Structure/Union member

diversity
Structure/Union member

key
Structure/Union member

next
Structure/Union member

ordinal
Structure/Union member

zero
Structure/Union member

class cle.backends.macho.structs.**Arm64e**
Bases: Union

named after the Union *Arm64e* from dyld MachOLoaded.h https://github.com/apple-opensource/dyld/blob/852.2/dyld3/MachOLoaded.h#L89-L103

authRebase: *dyld_chained_ptr_arm64e_auth_rebase*
Structure/Union member

authBind: *dyld_chained_ptr_arm64e_auth_bind*
Structure/Union member

rebase: *dyld_chained_ptr_arm64e_rebase*

Structure/Union member

bind: *dyld_chained_ptr_arm64e_bind*

Structure/Union member

bind24: *dyld_chained_ptr_arm64e_bind24*

Structure/Union member

authBind24: *dyld_chained_ptr_arm64e_auth_bind24*

Structure/Union member

property sign_extended_addend

property unpack_target

static check_valid_pointer_format(*pointer_format:* DyldChainedPtrFormats) → bool

helper to check if a pointer format is relevant for this :type pointer_format: *DyldChainedPtrFormats* :param pointer_format:

Return type
bool

Returns

Parameters
pointer_format (DyldChainedPtrFormats)

class cle.backends.macho.structs.**dyld_chained_ptr_64_rebase**

Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L153-L161

target: int

Structure/Union member

high8: int

Structure/Union member

next: int

Structure/Union member

bind: int

Structure/Union member

property unpackedTarget

class cle.backends.macho.structs.**dyld_chained_ptr_64_bind**

Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L189-L197

ordinal: int

Structure/Union member

addend: int

Structure/Union member

next: int

Structure/Union member

bind: *int*

Structure/Union member

class cle.backends.macho.structs.**Generic64**

Bases: Union

named after the Union *Generic64* from dyld MachOLoaded.h https://github.com/apple-opensource/dyld/blob/852.2/dyld3/MachOLoaded.h#L105-L111

rebase: *dyld_chained_ptr_64_rebase*

Structure/Union member

bind: *dyld_chained_ptr_64_bind*

Structure/Union member

static check_valid_pointer_format(*pointer_format:* DyldChainedPtrFormats) → bool

Return type
bool

Parameters
pointer_format (DyldChainedPtrFormats)

class cle.backends.macho.structs.**ChainedFixupPointerOnDisk**

Bases: Union

the ChainedFixupPointerOnDisk union from dyld MachOLoaded.h https://github.com/apple-opensource/dyld/blob/852.2/dyld3/MachOLoaded.h#L87-L141

generic64: *Generic64*

Structure/Union member

arm64e: *Arm64e*

Structure/Union member

isBind(*pointer_format:* DyldChainedPtrFormats) → tuple[int, int] | None

Port of ChainedFixupPointerOnDisk::isBind(uint16_t pointerFormat, uint32_t& bindOrdinal, int64_t& addend) https://github.com/apple-opensource/dyld/blob/852.2/dyld3/MachOLoaded.cpp#L1098-L1147 Returns None if not a bind (so *if struct.isBind()* works),

Return type
tuple[int, int] | None

Returns

Parameters
pointer_format (DyldChainedPtrFormats)

isRebase(*pointer_format:* DyldChainedPtrFormats, *preferredLoadAddress: int*) → int | None

port of ChainedFixupPointerOnDisk::isRebase(uint16_t pointerFormat, uint64_t preferedLoadAddress, uint64_t& targetRuntimeOffset) https://github.com/apple-opensource/dyld/blob/852.2/dyld3/MachOLoaded.cpp#L1046-L1096 :type pointer_format: *DyldChainedPtrFormats* :param pointer_format: :type preferredLoadAddress: int :param preferredLoadAddress: I think that's just the requested base address

Return type
int | None

Returns

Parameters

- **pointer_format** (DyldChainedPtrFormats)
- **preferredLoadAddress** (*int*)

class cle.backends.macho.structs.**DyldImportStruct**

Bases: *HelperStruct*

Meta Struct for the different kind of import structs and the fields they are all guaranteed to have

lib_ordinal: int

weak_import: bool

name_offset: int

static get_struct(*pointer:* DyldImportFormats) → *type*[*DyldImportStruct*]

Return type
type[*DyldImportStruct*]

Parameters
pointer (DyldImportFormats)

class cle.backends.macho.structs.**dyld_chained_import**

Bases: *DyldImportStruct*

Struct for symbol format DYLD_CHAINED_IMPORT

lib_ordinal: int

Structure/Union member

name_offset: int

Structure/Union member

weak_import: bool

Structure/Union member

class cle.backends.macho.structs.**dyld_chained_import_addend**

Bases: *DyldImportStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L264-L271

addend: int

Structure/Union member

lib_ordinal: int

Structure/Union member

name_offset: int

Structure/Union member

weak_import: bool

Structure/Union member

class cle.backends.macho.structs.**dyld_chained_import_addend64**

Bases: *DyldImportStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L273-L281

addend: int

Structure/Union member

lib_ordinal: int
Structure/Union member

name_offset: int
Structure/Union member

reserved
Structure/Union member

weak_import: bool
Structure/Union member

class cle.backends.macho.structs.**dyld_chained_fixups_header**
Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L36-L46

fixups_version: int
Structure/Union member

starts_offset: int
Structure/Union member

imports_offset: int
Structure/Union member

symbols_offset: int
Structure/Union member

imports_count: int
Structure/Union member

imports_format: *DyldImportFormats*
Structure/Union member

symbols_format: int
Structure/Union member

class cle.backends.macho.structs.**dyld_chained_starts_in_image**
Bases: Structure

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L48-L54

seg_count: int
Structure/Union member

seg_info_offset: Array
Structure/Union member

class cle.backends.macho.structs.**dyld_chained_starts_in_segment**
Bases: *HelperStruct*

https://github.com/apple-opensource/dyld/blob/852.2/include/mach-o/fixup-chains.h#L56-L72

page_size: int
Structure/Union member

segment_offset: int
Structure/Union member

max_valid_pointer: *int*

Structure/Union member

page_count: *int*

Structure/Union member

page_start: *int*

Structure/Union member

property pointer_format: *DyldChainedPtrFormats*

2.3.4 Binary Ninja

class cle.backends.binja.**BinjaSymbol**

Bases: *Symbol*

BINJA_FUNC_SYM_TYPES = []

BINJA_DATA_SYM_TYPES = []

BINJA_IMPORT_TYPES = []

__init__(*owner*, *sym*)

Not documenting this since if you try calling it, you're wrong.

class cle.backends.binja.**BinjaReloc**

Bases: *Relocation*

property value

class cle.backends.binja.**BinjaBin**

Bases: *Backend*

Get information from binaries using Binary Ninja. Basing this on idabin.py, but will try to be more complete. TODO: add more features as Binary Ninja's feature set improves

is_default = True

BINJA_ARCH_MAP = {'aarch64': <Arch AARCH64 (LE)>, 'armv7': <Arch ARMEL (LE)>, 'armv7eb': <Arch ARMEL (BE)>, 'mips32': <Arch MIPS32 (BE)>, 'mipsel32': <Arch MIPS32 (LE)>, 'ppc': <Arch PPC32 (BE)>, 'ppc_le': <Arch PPC32 (LE)>, 'thumb2': <Arch ARMEL (LE)>, 'thumb2eb': <Arch ARMEL (BE)>, 'x86': <Arch X86 (LE)>, 'x86_64': <Arch AMD64 (LE)>}

__init__(*binary*, **args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

in_which_segment(*addr*)

Return the segment name at address *addr*.

get_symbol_addr(*sym*)

Get the address of the symbol *sym* from IDA.

Returns
An address.

function_name(*addr*)

Return the function name at address *addr*.

property min_addr

this is probably not "right")

Type
Get the min address of the binary. (note

property max_addr

Get the max address of the binary.

property entry

get_strings()

Extract strings from binary (Binary Ninja).

Returns
An array of strings.

set_got_entry(*name*, *newaddr*)

Resolve import *name* with address *newaddr*. That is, update the GOT entry for *name* with *newaddr*.

close()

Release the BinaryView we created in __init__ :return: None

2.3.5 Blob

class cle.backends.blob.**Blob**

Bases: *Backend*

Representation of a binary blob, i.e. an executable in an unknown file format.

is_default = True

__init__(**args*, *offset=None*, *segments=None*, ***kwargs*)

Parameters

- **arch** – (required) an archinfo.Arch for the binary blob.
- **offset** – Skip this many bytes from the beginning of the file.
- **segments** – List of tuples describing how to map data into memory. Tuples are of `(file_offset, mem_addr, size)`.

You can't specify both `offset` and `segments`.

classmethod is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

property min_addr

This returns the lowest virtual address contained in any loaded segment of the binary.

property max_addr

This returns the highest virtual address contained in any loaded segment of the binary.

function_name(*addr*)

Blobs don't support function names.

contains_addr(*addr*)

Is *addr* in one of the binary's segments/sections we have loaded? (i.e. is it mapped into memory ?)

in_which_segment(*addr*)

Blobs don't support segments.

classmethod check_compatibility(*spec*, *obj*)

Performs a minimal static load of `spec` and returns whether it's compatible with other_obj

2.3.6 COFF

Basic MS COFF object loader based on https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

class cle.backends.coff.**IMAGE_FILE_MACHINE**

Bases: IntEnum

Machine Types

I386 = 332

AMD64 = 34404

__new__(*value*)

class cle.backends.coff.**CoffFileHeader**

Bases: Structure

COFF File Header

Characteristics

Structure/Union member

Machine

Structure/Union member

NumberOfSections

Structure/Union member

NumberOfSymbols

Structure/Union member

PointerToSymbolTable

Structure/Union member

SizeOfOptionalHeader

Structure/Union member

TimeDateStamp

Structure/Union member

class cle.backends.coff.**IMAGE_SCN**

Bases: IntFlag

Section Flags (Characteristics field)

MEM_EXECUTE = 536870912

MEM_READ = 1073741824

MEM_WRITE = 2147483648

CNT_UNINITIALIZED_DATA = 128

__new__(*value*)

class cle.backends.coff.**CoffSectionTableEntry**

Bases: Structure

COFF Section Header

Characteristics

Structure/Union member

Name

Structure/Union member

NumberOfLinenumbers

Structure/Union member

NumberOfRelocations

Structure/Union member

PointerToLinenumbers

Structure/Union member

PointerToRawData

Structure/Union member

PointerToRelocations

Structure/Union member

SizeOfRawData

Structure/Union member

VirtualAddress

Structure/Union member

VirtualSize

Structure/Union member

class cle.backends.coff.**IMAGE_SYM_CLASS**

Bases: IntEnum

Symbol Storage Class

EXTERNAL = 2

STATIC = 3

LABEL = 6

FUNCTION = 101

__new__(*value*)

class cle.backends.coff.**CoffSymbolTableEntry**

Bases: Structure

COFF Symbol Table Entry

Name

Structure/Union member

NumberOfAuxSymbols

Structure/Union member

SectionNumber

Structure/Union member

StorageClass

Structure/Union member

Type

Structure/Union member

Value

Structure/Union member

class cle.backends.coff.**IMAGE_REL_I386**

Bases: IntEnum

i386 Relocation Types

DIR32 = 6

DIR32NB = 7

REL32 = 20

SECTION = 10

SECREL = 11

__new__(*value*)

class cle.backends.coff.**IMAGE_REL_AMD64**

Bases: IntEnum

AMD64 Relocation Types

ADDR64 = 1

ADDR32NB = 3

REL32 = 4

SECTION = 10

SECREL = 11

__new__(*value*)

class cle.backends.coff.**CoffRelocationTableEntry**

Bases: Structure

COFF Relocations

SymbolTableIndex

Structure/Union member

Type

Structure/Union member

VirtualAddress

Structure/Union member

class cle.backends.coff.**CoffParser**

Bases: object

Parses COFF object files.

header: *CoffFileHeader*

sections: list[*CoffSectionTableEntry*]

relocations: list[list[*CoffRelocationTableEntry*]]

symbols: list[*CoffSymbolTableEntry*]

idx_to_symbol_name: dict[int, str]

symbol_name_to_idx: dict[str, int]

__init__(*data: bytes*)

Parameters
data (*bytes*)

data: bytes

get_symbol_name(*symbol_idx: int*, *true_name: bool = False*) → str

Return type
str

Parameters

- **symbol_idx** (*int*)
- **true_name** (*bool*)

get_section_name(*section_idx: int*) → str

Return type
str

Parameters
section_idx (*int*)

class cle.backends.coff.**CoffSection**

Bases: *Section*

Section of the COFF object.

__init__(*name: str*, *file_offset: int*, *file_size: int*, *virtual_addr: int*, *virtual_size: int*, *coff_sec:* CoffSectionTableEntry)

Parameters

- **name** (str) – The name of the section
- **offset** (int) – The offset into the binary file this section begins
- **vaddr** (int) – The address in virtual memory this section begins
- **size** (int) – How large this section is
- **file_offset** (int)
- **file_size** (int)
- **virtual_addr** (int)
- **virtual_size** (int)
- **coff_sec** (CoffSectionTableEntry)

property is_readable

Whether this section has read permissions

property is_writable

Whether this section has write permissions

property is_executable

Whether this section has execute permissions

property only_contains_uninitialized_data

Whether this section is initialized to zero after the executable is loaded.

class cle.backends.coff.**CoffRelocation**

Bases: *Relocation*

Relocation for a COFF object.

PACK_FORMAT = '<i'

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

class cle.backends.coff.**CoffRelocationREL32**

Bases: *CoffRelocation*

Relocation for IMAGE_REL_*_REL32

property value

class cle.backends.coff.**CoffRelocationDIR32**

Bases: *CoffRelocation*

Relocation for IMAGE_REL_*_DIR32

property value

class cle.backends.coff.**CoffRelocationDIR32NB**

Bases: *CoffRelocation*

Relocation for IMAGE_REL_*_DIR32

property value

class cle.backends.coff.**CoffRelocationADDR32NB**

Bases: *CoffRelocation*

Relocation for IMAGE_REL_AMD64_ADDR32NB

PACK_FORMAT = '<I'

property value: int

class cle.backends.coff.**CoffRelocationADDR64**

Bases: *CoffRelocation*

Relocation for IMAGE_REL_AMD64_ADDR64

PACK_FORMAT = '<Q'

property value

class cle.backends.coff.**CoffRelocationSECTION**

Bases: *CoffRelocation*

Relocation for IMAGE_REL_*_SECTION

PACK_FORMAT = '<H'

property value

class cle.backends.coff.**CoffRelocationSECREL**

Bases: *CoffRelocation*

Relocation for IMAGE_REL_*_SECREL

PACK_FORMAT = '<I'

property value

class cle.backends.coff.**Coff**

Bases: *Backend*

COFF object loader.

is_default = True

__init__(**args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

classmethod is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

get_symbol(*name: str*, *produce_extern_symbols: bool = False*) → *Symbol* | None

Stub function. Implement to find the symbol with name *name*.

Return type
Symbol | None

Parameters

- **name** (*str*)
- **produce_extern_symbols** (*bool*)

2.3.7 CGC

class cle.backends.cgc.**CGC**

Bases: *ELF*

Backend to support the CGC elf format used by the Cyber Grand Challenge competition.

See : https://github.com/CyberGrandChallenge/libcgcef/blob/master/cgc_executable_format.md

is_default = True

__init__(*binary*, *binary_stream*, **args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

supported_filetypes = ['cgc']

class cle.backends.cgc.**BackedCGC**

Bases: *CGC*

This is a backend for CGC executables that allows user provide a memory backer and a register backer as the initial state of the running binary.

is_default = True

__init__(**args*, *memory_backer=None*, *register_backer=None*, *writes_backer=None*, *permissions_map=None*, *current_allocation_base=None*, ***kwargs*)

Parameters

- **path** – File path to CGC executable.
- **memory_backer** – A dict of memory content, with beginning address of each segment as key and actual memory content as data.
- **register_backer** – A dict of all register contents. EIP will be used as the entry point of this executable.
- **permissions_map** – A dict of memory region to permission flags
- **current_allocation_base** – An integer representing the current address of the top of the CGC heap.

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

property threads

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This property should contain a list of names for these threads, which should be unique.

thread_registers(*thread=None*)

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This method should return the register file for a given thread (as named in `Backend.threads`) as a dict mapping register names (as seen in archinfo) to numbers. If the thread is not specified, it should return the context for a "default" thread. If there are no threads, it should return an empty dict.

class cle.backends.cgc.cgc.**CGC**

Bases: *ELF*

Backend to support the CGC elf format used by the Cyber Grand Challenge competition.

See : https://github.com/CyberGrandChallenge/libcgcef/blob/master/cgc_executable_format.md

is_default = True

__init__(*binary*, *binary_stream*, **args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

supported_filetypes = ['cgc']

class cle.backends.cgc.backedcgc.**FakeSegment**

Bases: *Segment*

__init__(*start*, *size*)

class cle.backends.cgc.backedcgc.**BackedCGC**

Bases: *CGC*

This is a backend for CGC executables that allows user provide a memory backer and a register backer as the initial state of the running binary.

is_default = True

__init__(**args*, *memory_backer=None*, *register_backer=None*, *writes_backer=None*, *permissions_map=None*, *current_allocation_base=None*, ***kwargs*)

Parameters

- **path** – File path to CGC executable.
- **memory_backer** – A dict of memory content, with beginning address of each segment as key and actual memory content as data.
- **register_backer** – A dict of all register contents. EIP will be used as the entry point of this executable.

- **permissions_map** – A dict of memory region to permission flags
- **current_allocation_base** – An integer representing the current address of the top of the CGC heap.

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

property threads

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This property should contain a list of names for these threads, which should be unique.

thread_registers(*thread=None*)

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This method should return the register file for a given thread (as named in `Backend.threads`) as a dict mapping register names (as seen in archinfo) to numbers. If the thread is not specified, it should return the context for a "default" thread. If there are no threads, it should return an empty dict.

2.3.8 IHex

class cle.backends.ihex.**Hex**

Bases: *Backend*

A loader for Intel Hex Objects See https://en.wikipedia.org/wiki/Intel_HEX

is_default = True

static parse_record(*line*)

static coalesce_regions(*regions*)

__init__(**args*, *ignore_missing_arch: bool = False*, *extra_regions: list[dict[str, int | bool]] = None*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable
- **ignore_missing_arch** (*bool*)
- **extra_regions** (*list[dict[str, int | bool]]*)

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

2.3.9 Java

Constant values for lifecycle of Apk.

class cle.backends.java.apk.**Apk**

Bases: *Soot*

Backend for lifting Apk's to Soot.

is_default = True

__init__(*apk_path*, *binary_stream*, *entry_point=None*, *entry_point_params=()*, *android_sdk=None*, *supported_jni_archs=None*, *jni_libs=None*, *jni_libs_ld_path=None*, ***options*)

Parameters

- **apk_path** – Path to APK.
- **android_sdk** – Path to Android SDK folder (e.g. "/home/angr/android/platforms")

The following parameters are optional

Parameters

- **entry_point** – Fully qualified name of method that should be used as the entry point.
- **supported_jni_archs** – List of supported JNI architectures (ABIs) in descending order of preference.
- **jni_libs** – Name(s) of JNI libs to load (if any). If not specified, we try to extract JNI libs from the APK.
- **jni_libs_ld_path** – Path(s) where to find libs defined by param jni_libs. Note: Directory of the APK is added by default.

get_callbacks(*class_name: str*, *callback_names: list[str]*) → list[None]

Get callback methods from the name of callback methods.

Parameters

- **class_name** (str) – Name of the class.
- **callback_names** (list[str]) – Name list of the callbacks.

Returns
The method object which is callback.

Return type
list[None]

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

class cle.backends.java.jar.**Jar**

Bases: *Soot*

Backend for lifting JARs to Soot.

is_default = True

__init__(*jar_path*, *binary_stream*, *entry_point=None*, *entry_point_params=('java.lang.String[]',)*, *jni_libs=None*, *jni_libs_ld_path=None*, ***kwargs*)

Parameters
jar_path – Path to JAR.

The following parameters are optional

Parameters

- **entry_point** – Fully qualified name of method that should be used as the entry point. If not specified, we try to parse it from the manifest.
- **additional_jars** – Additional JARs.
- **additional_jar_roots** – Additional JAR roots.

- **jni_libs** – Name(s) of JNI libs to load (if any).
- **jni_libs_ld_path** – Path(s) where to find libs defined by param jni_libs. Note: Directory of the JAR is added by default.

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

get_manifest(*binary_path=None*)

Load the MANIFEST.MF file

Returns
A dict of meta info

Return type
dict

class cle.backends.java.soot.**Soot**

Bases: *Backend*

The basis backend for lifting and loading bytecode from JARs and APKs to Soot IR.

Note that self.min_addr will be 0 and self.max_addr will be 1. Hopefully no other object will be mapped at address 0.

__init__(**args*, *entry_point=None*, *entry_point_params=()*, *input_format=None*, *additional_jars=None*, *additional_jar_roots=None*, *jni_libs_ld_path=None*, *jni_libs=None*, *android_sdk=None*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

property max_addr

This returns the highest virtual address contained in any loaded segment of the binary.

property entry

property classes

get_soot_class(*cls_name*, *none_if_missing=False*)

Get Soot class object.

Parameters
cls_name (*str*) – Name of the class.

Returns
The class object.

Return type
pysoot.soot.SootClass

get_soot_method(*thing*, *class_name=None*, *params=()*, *none_if_missing=False*)

Get Soot method object.

Parameters

- **thing** – Descriptor or the method, or name of the method.

- **class_name** (*str*) – Name of the class. If not specified, class name can be parsed from method_name.

Returns
Soot method that satisfy the criteria.

property main_methods

Find all Main methods in this binary.

Returns
All main methods in each class.

Return type
iterator

static is_zip_archive(*stream*)

2.3.10 Minidump

exception cle.backends.minidump.**MinidumpMissingStreamError**

Bases: Exception

__init__(*stream*, *message=None*)

class cle.backends.minidump.**Minidump**

Bases: *Backend*

is_default = True

__init__(**args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

close()

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

property threads

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This property should contain a list of names for these threads, which should be unique.

thread_registers(*thread=None*)

If this backend represents a dump of a running program, it may contain one or more thread contexts, i.e. register files. This method should return the register file for a given thread (as named in `Backend.threads`) as a dict mapping register names (as seen in archinfo) to numbers. If the thread is not specified, it should return the context for a "default" thread. If there are no threads, it should return an empty dict.

get_thread_registers_by_id(*thread_id*)

2.3.11 Static Archive

class cle.backends.static_archive.**StaticArchive**

Bases: *Backend*

classmethod is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

is_default = True

is_outer = True

__init__(**args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

2.3.12 UEFI Firmware

exception cle.backends.uefi_firmware.**UefiDriverLoadError**

Bases: Exception

This error is raised (and caught internally) if the data contained in the UEFI entity tree doesn't make sense.

class cle.backends.uefi_firmware.**UefiFirmware**

Bases: *Backend*

A UEFI firmware blob loader. Support is provided by the `uefi_firmware` package.

is_default = True

classmethod is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

__init__(**args*, ***kwargs*) → None

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

Return type
None

class cle.backends.uefi_firmware.**UefiModulePending**

Bases: object

A worklist entry for the UEFI firmware loader.

name: str | None = None

pe_image: bytes | None = None

te_image: bytes | None = None

build(*parent:* UefiFirmware, *guid:* *UUID*) → *UefiModuleMixin*

Return type
UefiModuleMixin

Parameters

- **parent** (UefiFirmware)
- **guid** (*UUID*)

__init__(*name:* *str* | *None* = *None*, *pe_image:* *bytes* | *None* = *None*, *te_image:* *bytes* | *None* = *None*) → None

Parameters

- **name** (*str | None*)
- **pe_image** (*bytes | None*)
- **te_image** (*bytes | None*)

Return type
None

class cle.backends.uefi_firmware.**UefiModuleMixin**

Bases: *Backend*

A mixin to make other kinds of backends load as UEFI modules.

__init__(**args*, *guid:* *UUID*, *name:* *str* | *None*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable
- **guid** (*UUID*)
- **name** (*str | None*)

class cle.backends.uefi_firmware.**UefiPE**

Bases: *UefiModuleMixin*, *PE*

A PE file contained in a UEFI image.

class cle.backends.uefi_firmware.**UefiTE**

Bases: *UefiModuleMixin*, *TE*

A TE file contained in a UEFI image.

class cle.backends.te.**HeaderType**

Bases: tuple

HeaderType(signature, machine, number_of_sections, subsystem, stripped_size, address_of_entry_point, base_of_code, image_base, data_directory_0_virtual_address, data_directory_0_size, data_directory_1_virtual_address, data_directory_1_size)

static __new__(*_cls*, *signature*, *machine*, *number_of_sections*, *subsystem*, *stripped_size*, *address_of_entry_point*, *base_of_code*, *image_base*, *data_directory_0_virtual_address*, *data_directory_0_size*, *data_directory_1_virtual_address*, *data_directory_1_size*)

Create new instance of HeaderType(signature, machine, number_of_sections, subsystem, stripped_size, address_of_entry_point, base_of_code, image_base, data_directory_0_virtual_address, data_directory_0_size, data_directory_1_virtual_address, data_directory_1_size)

address_of_entry_point

Alias for field number 5

base_of_code

Alias for field number 6

data_directory_0_size

Alias for field number 9

data_directory_0_virtual_address

Alias for field number 8

data_directory_1_size

Alias for field number 11

data_directory_1_virtual_address

Alias for field number 10

image_base

Alias for field number 7

machine

Alias for field number 1

number_of_sections

Alias for field number 2

signature

Alias for field number 0

stripped_size

Alias for field number 4

subsystem

Alias for field number 3

class cle.backends.te.SectionHeaderType

Bases: tuple

SectionHeaderType(section_name, physical_address_virtual_size, virtual_address, size_of_raw_data, pointer_to_raw_data, pointer_to_relocations, pointer_to_line_numbers, number_of_relocations, number_of_line_numbers, characteristics)

static __new__(*_cls*, *section_name*, *physical_address_virtual_size*, *virtual_address*, *size_of_raw_data*, *pointer_to_raw_data*, *pointer_to_relocations*, *pointer_to_line_numbers*, *number_of_relocations*, *number_of_line_numbers*, *characteristics*)

Create new instance of SectionHeaderType(section_name, physical_address_virtual_size, virtual_address, size_of_raw_data, pointer_to_raw_data, pointer_to_relocations, pointer_to_line_numbers, number_of_relocations, number_of_line_numbers, characteristics)

characteristics
Alias for field number 9

number_of_line_numbers
Alias for field number 8

number_of_relocations
Alias for field number 7

physical_address_virtual_size
Alias for field number 1

pointer_to_line_numbers
Alias for field number 6

pointer_to_raw_data
Alias for field number 4

pointer_to_relocations
Alias for field number 5

section_name
Alias for field number 0

size_of_raw_data
Alias for field number 3

virtual_address
Alias for field number 2

class cle.backends.te.**TE**
Bases: *Backend*

A "Terse Executable" format image, commonly used as part of UEFI firmware drivers.

is_default = True

classmethod is_compatible(*stream*)
Determine quickly whether this backend can load an object from this stream

__init__(**args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

2.3.13 Xbox Executable

class cle.backends.xbe.**XBESection**
Bases: *Section*

__init__(*name*, *file_offset*, *file_size*, *virtual_addr*, *virtual_size*, *xbe_sec*)

Parameters

- **name** (*str*) – The name of the section

- **offset** (*int*) – The offset into the binary file this section begins
- **vaddr** (*int*) – The address in virtual memory this section begins
- **size** (*int*) – How large this section is

property is_readable

Whether this section has read permissions

property is_writable

Whether this section has write permissions

property is_executable

Whether this section has execute permissions

property only_contains_uninitialized_data

We load every section in, they're all initialized

class cle.backends.xbe.**XBE**

Bases: *Backend*

The main loader class for statically loading XBE executables.

is_default = True

__init__(**args*, ***kwargs*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

close()

static is_compatible(*stream*)

Determine quickly whether this backend can load an object from this stream

property min_addr

This returns the lowest virtual address contained in any loaded segment of the binary.

property max_addr

This returns the highest virtual address contained in any loaded segment of the binary.

classmethod check_compatibility(*spec*, *obj*)

Performs a minimal static load of `spec` and returns whether it's compatible with other_obj

2.4 Relocations

CLE's loader implements program relocations. If you would like to add support for more relocations, you can do so by subclassing the `Relocation` class and overriding any relevant methods or properties. Then, add or uncomment the appropriate line in the relocations_table dict at the bottom of the file. Look at the existing versions for details.

class cle.backends.relocation.**Relocation**

Bases: `object`

A representation of a relocation in a binary file. Smart enough to relocate itself.

Variables

- **owner** – The binary this relocation was originally found in, as a cle object
- **symbol** – The Symbol object this relocation refers to
- **relative_addr** – The address in owner this relocation would like to write to
- **resolvedby** (*Symbol* | None) – If the symbol this relocation refers to is an import symbol and that import has been resolved, this attribute holds the symbol from a different binary that was used to resolve the import.
- **resolved** (bool) – Whether the application of this relocation was successful

__init__(*owner: Backend*, *symbol: Symbol | None*, *relative_addr: int*)

Parameters

- **owner** (Backend)
- **symbol** (*Symbol | None*)
- **relative_addr** (*int*)

resolvedby: *Symbol* | None

resolved: bool

AUTO_HANDLE_NONE = False

resolve_symbol(*solist: list[Any]*, *thumb=False*, *extern_object=None*, ***kwargs*)

Parameters
solist (*list[Any]*)

resolve(*obj*, *extern_object=None*)

property rebased_addr

The address in the global memory space this relocation would like to write to

property linked_addr

property dest_addr

property value: int

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

property owner_obj

cle.backends.elf.relocation.**get_relocation**(*arch*, *r_type*)

class cle.backends.elf.relocation.elfreloc.**ELFReloc**

Bases: *Relocation*

__init__(*owner*, *symbol*, *relative_addr*, *addend=None*)

property addend

property value

class cle.backends.elf.relocation.generic.**GenericTLSDoffsetReloc**

Bases: *ELFReloc*

property value

resolve_symbol(*solist*, ***kwargs*)

class cle.backends.elf.relocation.generic.**GenericTLSOffsetReloc**

Bases: *ELFReloc*

AUTO_HANDLE_NONE = True

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

class cle.backends.elf.relocation.generic.**GenericTLSDescriptorReloc**

Bases: *ELFReloc*

RESOLVER_ADDR: int = NotImplemented

AUTO_HANDLE_NONE = True

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

class cle.backends.elf.relocation.generic.**GenericTLSModIdReloc**

Bases: *ELFReloc*

AUTO_HANDLE_NONE = True

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

class cle.backends.elf.relocation.generic.**GenericIRelativeReloc**

Bases: *ELFReloc*

AUTO_HANDLE_NONE = True

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

class cle.backends.elf.relocation.generic.**GenericAbsoluteAddendReloc**

Bases: *ELFReloc*

property value

class cle.backends.elf.relocation.generic.**GenericPCRelativeAddendReloc**

Bases: *ELFReloc*

property value

class cle.backends.elf.relocation.generic.**GenericJumpslotReloc**

Bases: *ELFReloc*

property value

class cle.backends.elf.relocation.generic.**GenericRelativeReloc**

Bases: *ELFReloc*

AUTO_HANDLE_NONE = True

property value

class cle.backends.elf.relocation.generic.**GenericAbsoluteReloc**

Bases: *ELFReloc*

property value

class cle.backends.elf.relocation.generic.**GenericCopyReloc**

Bases: *ELFReloc*

resolve_symbol(*solist*, ***kwargs*)

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

class cle.backends.elf.relocation.generic.**MipsGlobalReloc**

Bases: *GenericAbsoluteReloc*

class cle.backends.elf.relocation.generic.**MipsLocalReloc**

Bases: *ELFReloc*

AUTO_HANDLE_NONE = True

resolve_symbol(*solist*, ***kwargs*)

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

class cle.backends.elf.relocation.generic.**RelocTruncate32Mixin**

Bases: object

A mix-in class for relocations that cover a 32-bit field regardless of the architecture's address word length.

check_zero_extend = False

check_sign_extend = False

relocate()

class cle.backends.elf.relocation.generic.**RelocGOTMixin**

Bases: object

A mix-in class which will cause the symbol to be resolved to a pointer to the symbol instead of the symbol

resolve(*symbol*, *extern_object=None*)

Relocation types for PowerPC 32-bit architecture.

Reference: http://refspecs.linux-foundation.org/elf/elfspec_ppc.pdf page 4-18

Only relocations 1-37 are described in the document. The rest are from the GNU binutils source code. See include/elf/ppc.h in the binutils source code. Relocation types for PPC64.

Reference: http://refspecs.linuxfoundation.org/ELF/ppc64/PPC-elf64abi-1.9.pdf pages 57-59 Relocation types for i386.

Reference: https://github.com/hjl-tools/x86-psABI/wiki/intel386-psABI-1.1.pdf page 36 Relocations for amd64/x86_64

Reference: https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build page 73 Relocation types for MIPS 32-bit.

Reference: https://refspecs.linuxfoundation.org/elf/mipsabi.pdf page 4-19

The main document is old and does not contain all the relocation types. I could not find a more recent document, so I had to rely on the source code of GNU binutils for all relocations that are not in the main document. See include/elf/mips.h in the binutils source code. Relocation types for ARM.

Reference: https://github.com/ARM-software/abi-aa/blob/main/aaelf32/aaelf32.rst#relocation-codes Relocations for AARCH64

Reference: https://github.com/ARM-software/abi-aa/blob/main/aaelf64/aaelf64.rst#relocation Relocation types for the S390X architecture.

Reference: https://github.com/IBM/s390x-abi/releases/download/v1.6.1/lzsabi_s390x.pdf pages 51-52

cle.backends.pe.relocation.**get_relocation**(*arch*, *r_type*)

class cle.backends.pe.relocation.pereloc.**PEReloc**

Bases: *Relocation*

AUTO_HANDLE_NONE = True

__init__(*owner*, *symbol*, *addr*, *resolvewith=None*)

resolve_symbol(*solist*, *bypass_compatibility=False*, *extern_object=None*, ***kwargs*)

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

property value

property is_base_reloc

These relocations are ignored by the linker if the executable is loaded at its preferred base address. There is no associated symbol with base relocations.

property is_import

class cle.backends.pe.relocation.generic.**DllImport**

Bases: *PEReloc*

There's nothing special to be done for DLL imports but this class provides a unique name to the relocation type.

class cle.backends.pe.relocation.generic.**IMAGE_REL_BASED_ABSOLUTE**

Bases: *PEReloc*

relocate()

Applies this relocation. Will make changes to the memory object of the object it came from.

This implementation is a generic version that can be overridden in subclasses.

class cle.backends.pe.relocation.generic.**IMAGE_REL_BASED_HIGHADJ**

Bases: *PEReloc*

__init__(*owner*, *addr*, *next_rva*)

property value

In all the other cases, we can ignore the relocation difference part of the calculation because we simply use to_mva() to get our rebased address. In this case, however, we have to adjust the un-rebased address first.

class cle.backends.pe.relocation.generic.**IMAGE_REL_BASED_HIGHLOW**

Bases: *PEReloc*

property value

class cle.backends.pe.relocation.generic.**IMAGE_REL_BASED_DIR64**

Bases: *PEReloc*

property value

class cle.backends.pe.relocation.generic.**IMAGE_REL_BASED_HIGH**

Bases: *PEReloc*

property value

class cle.backends.pe.relocation.generic.**IMAGE_REL_BASED_LOW**

Bases: *PEReloc*

property value

2.5 Thread-local storage

class cle.backends.tls.**ThreadManager**

Bases: object

This class tracks what data is thread-local and can generate thread initialization images

Most of the heavy lifting will be handled in a subclass

__init__(*loader*, *arch*, *max_modules=256*)

register_object(*obj*)

static initialization_image(*obj*) → bytes | None

Return type
bytes | None

new_thread(*insert=True*)

class cle.backends.tls.**InternalTLSRelocation**

Bases: *Relocation*

AUTO_HANDLE_NONE = True

__init__(*val*, *offset*, *owner*)

property value

class cle.backends.tls.**TLSObject**

Bases: *Backend*

__init__(*loader*, *arch*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

class cle.backends.tls.**ELFThreadManager**

Bases: *ThreadManager*

__init__(**args*, ***kwargs*)

register_object(*obj*)

class cle.backends.tls.**ELFCoreThreadManager**

Bases: *ThreadManager*

__init__(*loader*, *arch*, ***kwargs*)

new_thread(*insert=False*)

register_object(*obj*)

class cle.backends.tls.**PEThreadManager**

Bases: *ThreadManager*

register_object(*obj*)

class cle.backends.tls.**MinidumpThreadManager**

Bases: *ThreadManager*

__init__(*loader*, *arch*, ***kwargs*)

new_thread(*insert=False*)

register_object(*obj*)

class cle.backends.tls.tls_object.**ThreadManager**

Bases: `object`

This class tracks what data is thread-local and can generate thread initialization images

Most of the heavy lifting will be handled in a subclass

__init__(*loader*, *arch*, *max_modules=256*)

register_object(*obj*)

static initialization_image(*obj*) → bytes | None

Return type
`bytes` | `None`

new_thread(*insert=True*)

class cle.backends.tls.tls_object.**InternalTLSRelocation**

Bases: *Relocation*

AUTO_HANDLE_NONE = True

__init__(*val*, *offset*, *owner*)

property value

class cle.backends.tls.tls_object.**TLSObject**

Bases: *Backend*

__init__(*loader*, *arch*)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable

This module is used when parsing the Thread Local Storage of an ELF binary. It heavily uses the TLSArchInfo named-tuple from archinfo.

ELF TLS is implemented based on the following documents:

- https://www.uclibc.org/docs/tls.pdf
- https://www.uclibc.org/docs/tls-ppc.txt
- https://www.uclibc.org/docs/tls-ppc64.txt
- https://www.linux-mips.org/wiki/NPTL

cle.backends.tls.elf_tls.**roundup**(*val*, *to=16*)

class cle.backends.tls.elf_tls.**ELFThreadManager**

Bases: *ThreadManager*

__init__(**args*, ***kwargs*)

register_object(*obj*)

class cle.backends.tls.elf_tls.**ELFTLSObject**

Bases: *TLSObject*

__init__(*thread_manager:* ELFThreadManager)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable
- **thread_manager** (*ELFThreadManager*)

property thread_pointer

The thread pointer. This is a technical term that refers to a specific location in the TLS segment.

property user_thread_pointer

The thread pointer that is exported to the user

property max_addr

This returns the highest virtual address contained in any loaded segment of the binary.

get_addr(*module_id*, *offset*)

basically `__tls_get_addr`.

class cle.backends.tls.elf_tls.**ELFTLSObjectV1**

Bases: *ELFTLSObject*

class cle.backends.tls.elf_tls.**ELFTLSObjectV2**

Bases: *ELFTLSObject*

class cle.backends.tls.pe_tls.**PEThreadManager**

Bases: *ThreadManager*

register_object(*obj*)

class cle.backends.tls.pe_tls.**PETLSObject**

Bases: *TLSObject*

This class is used when parsing the Thread Local Storage of a PE binary. It represents both the TLS array and the TLS data area for a specific thread.

In memory the `PETLSObj` is laid out as follows:

```
+----------------------+----------------------------------------+
| TLS array            | TLS data area                          |
+----------------------+----------------------------------------+
```

A more detailed description of the TLS array and TLS data areas is given below.

TLS array

The TLS array is an array of addresses that points into the TLS data area. In memory it is laid out as follows:

```
+-----------+-----------+-----+-----------+
|  address  |  address  | ... |  address  |
+-----------+-----------+-----+-----------+
| index = 0 | index = 1 |     | index = n |
+-----------+-----------+-----+-----------+
```

The size of each address is architecture independent (e.g. on X86 it is 4 bytes). The number of addresses in the TLS array is equal to the number of modules that contain TLS data. At load time (i.e. in the `finalize` method), each module is assigned an index into the TLS array. The address of this module's TLS data area is then stored at this location in the array.

TLS data area

The TLS data area directly follows the TLS array and contains the actual TLS data for each module. In memory it is laid out as follows:

```
+----------+-----------+----------+-----------+-----+
| TLS data | zero fill | TLS data | zero fill | ... |
+----------+-----------+----------+-----------+-----+
|       module a       |       module b       | ... |
+-----------------------------------------------------+
```

The size of each module's TLS data area is variable and can be found in the module's `tls_data_size` property. The same applies to the zero fill. At load time (i.e in the `finalize` method), the initial TLS data values are copied into the TLS data area. Because a TLS index is also assigned to each module, we can access a module's TLS data area using this index into the TLS array to get the start address of the TLS data.

__init__(*thread_manager:* PEThreadManager)

Parameters

- **binary** – The path to the binary to load
- **binary_stream** – The open stream to this binary. The reference to this will be held until you call close.
- **is_main_bin** – Whether this binary should be loaded as the main executable
- **thread_manager** (PEThreadManager)

get_tls_data_addr(*tls_idx*)

Get the start address of a module's TLS data area via the module's TLS index.

From the PE/COFF spec:

> The code uses the TLS index and the TLS array location (multiplying the index by the word size and using it as an offset into the array) to get the address of the TLS data area for the given program and module.

property max_addr

This returns the highest virtual address contained in any loaded segment of the binary.

property thread_pointer

property user_thread_pointer

class cle.backends.tls.elfcore_tls.**ELFCoreThreadManager**

Bases: *ThreadManager*

__init__(*loader*, *arch*, ***kwargs*)

new_thread(*insert=False*)

register_object(*obj*)

class cle.backends.tls.elfcore_tls.**ELFCoreThread**

Bases: object

__init__(*loader*, *arch: Arch*, *threadinfo*)

Parameters
arch (*Arch*)

property dtv

get_addr(*module_id*, *offset*)

basically `__tls_get_addr`.

class cle.backends.tls.minidump_tls.**MinidumpThreadManager**

Bases: *ThreadManager*

__init__(*loader*, *arch*, ***kwargs*)

new_thread(*insert=False*)

register_object(*obj*)

class cle.backends.tls.minidump_tls.**MinidumpThread**

Bases: object

__init__(*loader*, *arch: Arch*, *registers*)

Parameters
arch (*Arch*)

get_tls_data_addr(*tls_idx*)

2.6 Errors

exception cle.errors.**CLEError**

Bases: Exception

Base class for errors raised by CLE.

exception cle.errors.**CLEUnknownFormatError**

Bases: *CLEError*

Error raised when CLE encounters an unknown executable file format.

exception cle.errors.**CLEFileNotFoundError**

Bases: *CLEError*

Error raised when a file does not exist.

exception cle.errors.**CLEInvalidBinaryError**

Bases: *CLEError*

Error raised when an executable file is invalid or corrupted.

exception cle.errors.**CLEOperationError**

Bases: *CLEError*

Error raised when a problem is encountered in the process of loading an executable.

exception cle.errors.**CLECompatibilityError**

Bases: *CLEError*

Error raised when loading an executable that is not currently supported by CLE.

exception cle.errors.**CLEMemoryError**

Bases: *CLEError*

Error raised when performing memory operations on unmapped addresses

2.7 Misc. Utilities

cle.gdb.**convert_info_sharedlibrary**(*fname*)

Convert a dump from gdb's `info sharedlibrary` command to a set of options that can be passed to CLE to replicate the address space from the gdb session

Parameters
fname – The name of a file containing the dump

Returns
A dict appropriate to be passed as `**kwargs` for `angr.Project` or `cle.Loader`

`cle.gdb.convert_info_proc_maps(fname)`

Convert a dump from gdb's `info proc maps` command to a set of options that can be passed to CLE to replicate the address space from the gdb session

Parameters
fname – The name of a file containing the dump

Returns
A dict appropriate to be passed as `**kwargs` for `angr.Project` or `cle.Loader`

`class cle.memory.ClemoryBase`

Bases: `object`

The base class of all Clemory classes.

`__init__(arch)`

`load(addr, n)`

`store(addr, data)`

`backers(addr=0)`

`find(data, search_min=None, search_max=None)` → Iterator[int]

Return type
`Iterator[int]`

`unpack(addr: int, fmt: str)` → tuple[Any, ...]

Use the `struct` module to unpack the data at address *addr* with the format *fmt*.

Return type
`tuple[Any, ...]`

Parameters

- **addr** (*int*)
- **fmt** (*str*)

`unpack_word(addr: int, size: int | None = None, signed: bool = False, endness: Endness | None = None)` → int

Use the `struct` module to unpack a single integer from the address *addr*.

You may override any of the attributes of the word being extracted:

Parameters

- **size** (`int | None`) – The size in bytes to pack/unpack. Defaults to wordsize (e.g. 4 bytes on a 32 bit architecture)
- **signed** (`bool`) – Whether the data should be extracted signed/unsigned. Default unsigned
- **endness** (`Endness | None`) – The endian to use in packing/unpacking. Defaults to memory endness
- **addr** (*int*)

Return type
`int`

load_null_terminated_bytes(*addr: int*, *max_size: int = 4096*) → bytes

Load a null-terminated string from memory at address *addr* with a maximum size of *max_size*. Useful

Return type
bytes

Parameters

- **addr** (*int*)
- **max_size** (*int*)

pack(*addr: int*, *fmt: str*, **data*)

Use the `struct` module to pack *data* into memory at address *addr* with the format *fmt*.

Parameters

- **addr** (*int*)
- **fmt** (*str*)

pack_word(*addr: int*, *data: int*, *size: int | None = None*, *signed: bool = False*, *endness: Endness | None = None*)

Use the `struct` module to pack a single integer *data* into memory at the address *addr*.

You may override any of the attributes of the word being packed:

Parameters

- **size** (int | None) – The size in bytes to pack/unpack. Defaults to wordsize (e.g. 4 bytes on a 32 bit architecture)
- **signed** (bool) – Whether the data should be extracted signed/unsigned. Default unsigned
- **endness** (Endness | None) – The endian to use in packing/unpacking. Defaults to memory endness
- **addr** (*int*)
- **data** (*int*)

read(*nbytes: int*)

The stream-like function that reads up to a number of bytes starting from the current position and updates the current position. Use with *seek()*.

Up to *nbytes* bytes will be read, halting at the beginning of the first unmapped region encountered.

Parameters
nbytes (*int*)

seek(*value: int*)

The stream-like function that sets the "file's" current position. Use with *read()*.

Parameters
value (int) – The position to seek to.

tell() → int

Return type
int

close()

class cle.memory.**Clemory**

Bases: *ClemoryBase*

An object representing a memory space.

Accesses can be made with [index] notation.

__init__(*arch: Arch*, *root: bool = False*)

Parameters

- **arch** (*Arch*)
- **root** (*bool*)

consecutive: bool

min_addr: int

max_addr: int

add_backer(*start: int*, *data: bytes | bytearray | memoryview | list[int] | Clemory | mmap*, *overwrite: bool = False*)

Adds a backer to the memory.

Parameters

- **start** (int) – The address where the backer should be loaded.
- **data** (bytes | bytearray | memoryview | list[int] | *Clemory* | mmap) – The backer itself. Can be either a bytestring or another *Clemory*.
- **overwrite** (bool) – If True and the range overlaps any existing backer, the existing backer will be split up and the overlapping part will be replaced with the new backer.

split_backer(*addr: int*)

Ensures that addr is the start of a backer, if it is backed.

Parameters
addr (*int*)

remove_backer(*start*)

backers(*addr=0*) → Iterator[tuple[int, bytearray | memoryview | mmap | list[int]]]

Iterate through each backer for this clemory and all its children, yielding tuples of `(start_addr, backer)` where each backer is a bytearray.

Parameters
addr – An optional starting address - all backers before and not including this address will be skipped.

Return type
Iterator[tuple[int, bytearray | memoryview | mmap | list[int]]]

load(*addr*, *n*)

Read up to *n* bytes at address *addr* in memory and return a bytes object.

Reading will stop at the beginning of the first unallocated region found, or when *n* bytes have been read.

store(*addr*, *data*)

Write bytes from *data* at address *addr*.

Note: If the store runs off the end of a backer and into unbacked space, this function will update the backer but also raise KeyError.

find(*data*, *search_min=None*, *search_max=None*) → Iterator[int]

Find all occurances of a bytestring in memory.

Parameters

- **data** (*bytes*) – The bytestring to search for
- **search_min** (*int*) – Optional: The first address to include as valid
- **search_max** (*int*) – Optional: The last address to include as valid

Return Iterator[int]
Iterates over addresses at which the bytestring occurs

Return type
Iterator[int]

class cle.memory.**ClemoryView**

Bases: *ClemoryBase*

A Clemory which presents a subset of another Clemory as an address space.

__init__(*backer*, *start*, *end*, *offset=0*)

Parameters

- **backer** – The parent clemory to use
- **start** – The address in the parent to start at
- **end** – The address in the parent to end at (exclusive)
- **offset** – Where the address space should start in this Clemory. Default 0.

backers(*addr=0*)

load(*addr*, *n*)

store(*addr*, *data*)

find(*data*, *search_min=None*, *search_max=None*) → Iterator[int]

Return type
Iterator[int]

class cle.memory.**ClemoryTranslator**

Bases: *ClemoryBase*

Uses a function to translate between address spaces when accessing a child clemory. Intended to be used only as a stream object.

__init__(*backer:* ClemoryBase, *func*)

Parameters
backer (ClemoryBase)

load(*addr*, *n*)

store(*addr*, *data*)

backers(*addr=0*)

find(*data*, *search_min=None*, *search_max=None*) → Iterator[int]

Return type
Iterator[int]

class cle.memory.**UninitializedClemory**

Bases: *Clemory*

A special kind of Clemory that acts as a placeholder for uninitialized and invalid memory. This is needed for the PAGEZERO segment for MachO binaries, which is 4GB worth of memory This does _not_ handle data being written to it, this is only for uninitialized memory that is technically occupied but should never be accessed

__init__(*arch*, *size*)

max_addr: int

add_backer(*start*, *data*, *overwrite=False*)

Adds a backer to the memory.

Parameters

- **start** – The address where the backer should be loaded.
- **data** – The backer itself. Can be either a bytestring or another *Clemory*.
- **overwrite** – If True and the range overlaps any existing backer, the existing backer will be split up and the overlapping part will be replaced with the new backer.

split_backer(*addr*)

Ensures that addr is the start of a backer, if it is backed.

remove_backer(*start*)

backers(*addr=0*)

Technically this object has no real backer We could create a fake backer on demand, but that would be a waste of memory, and code like the function prolog discovery for MachO binaries would search 4GB worth of nullbytes for a prolog, which is a waste of time Instead we just return an empty byte array, which seems to pass the test cases :type addr: :param addr: :return:

load(*addr*, *n*)

Read up to *n* bytes at address *addr* in memory and return a bytes object.

Reading will stop at the beginning of the first unallocated region found, or when *n* bytes have been read.

store(*addr*, *data*)

Write bytes from *data* at address *addr*.

Note: If the store runs off the end of a backer and into unbacked space, this function will update the backer but also raise KeyError.

find(*data*, *search_min=None*, *search_max=None*) → Iterator[int]

The memory has no value, so matter what is searched for, it won't be found.

Return type
Iterator[int]

consecutive: bool

min_addr: int

class cle.patched_stream.**PatchedStream**

Bases: object

An object that wraps a readable stream, performing passthroughs on seek and read operations, except to make it seem like the data has actually been patched by the given patches.

__init__(*stream*, *patches*)

Parameters

- **stream** – The stream to patch
- **patches** – A list of tuples of (addr, patch data)

read(**args*, ***kwargs*)

seek(**args*, ***kwargs*)

tell()

close()

class cle.address_translator.**AddressTranslator**

Bases: object

__init__(*rva*, *owner*)

Parameters

- **rva** (*int*) – virtual address relative to owner's object image base
- **owner** (*cle.Backend*) – The object owner address relates to

classmethod from_lva(*lva*, *owner*)

Loads address translator with LVA

classmethod from_mva(*mva*, *owner*)

Loads address translator with MVA

classmethod from_rva(*rva*, *owner*)

Loads address translator with RVA

classmethod from_raw(*raw*, *owner*)

Loads address translator with RAW address

classmethod from_linked_va(*lva*, *owner*)

Loads address translator with LVA

classmethod from_va(*mva*, *owner*)

Loads address translator with MVA

classmethod from_mapped_va(*mva*, *owner*)

Loads address translator with MVA

classmethod from_relative_va(*rva*, *owner*)

Loads address translator with RVA

to_lva()

VA -> LVA :rtype: int

to_mva()
RVA -> MVA :rtype: int

to_rva()
RVA -> RVA :rtype: int

to_raw()
RVA -> RAW :rtype: int

to_linked_va()
VA -> LVA :rtype: int

to_va()
RVA -> MVA :rtype: int

to_mapped_va()
RVA -> MVA :rtype: int

to_relative_va()
RVA -> RVA :rtype: int

cle.address_translator.**AT**
alias of *AddressTranslator*

cle.utils.**ALIGN_DOWN**(*base*, *size*)

cle.utils.**ALIGN_UP**(*base*, *size*)

cle.utils.**get_mmaped_data**(*stream*, *offset*, *length*, *page_size*)

cle.utils.**stream_or_path**(*obj*, *perms='rb'*)

cle.utils.**key_bisect_floor_key**(*lst*, *key*, *lo=0*, *hi=None*, *keyfunc=<function <lambda>>*)

cle.utils.**key_bisect_find**(*lst*, *item*, *lo=0*, *hi=None*, *keyfunc=<function <lambda>>*)

cle.utils.**key_bisect_insort_left**(*lst*, *item*, *lo=0*, *hi=None*, *keyfunc=<function <lambda>>*)

cle.utils.**key_bisect_insort_right**(*lst*, *item*, *lo=0*, *hi=None*, *keyfunc=<function <lambda>>*)

cle.utils.**get_text_offset**(*path*)
Offset of text section in the binary.

cle.utils.**extract_null_terminated_bytestr**(*data: bytes | bytearray*, *offset: int = 0*, *sentinel_value: bytes = b'\\x00'*) → bytes
Return an exclusive null-terminated sequence of bytes at *offset* in *data*.

Return type
bytes

Parameters

- **data** (*bytes | bytearray*)
- **offset** (*int*)
- **sentinel_value** (*bytes*)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z