FIDL Documentation

Release 1.0

FLARE's OTF

May 22, 2020

Documentation

```
    FLARE IDA DECOMPILER LIBRARY
```

CHAPTER 1

What?

FIDL is a library wrapping the *Hex-Rays* API, allowing you to leverage Hex-Rays decompiler power to perform better binary analysis.

CHAPTER 2

Why?

The *Hex-Rays* API is notoriously difficult to use. If you have ever tried to do so, you know. If not, then you are lucky :)

CHAPTER 3

Getting help

Having trouble? We'd like to help!

- Looking for specific information? Try the genindex or modindex.
- Ask or search questions in StackOverflow using the FIDL tag.
- Report bugs with FIDL in our issue tracker.

CHAPTER 4

First steps

4.1 Installation

FIDL is just another Python package living within your local Python installation. There are two ways to install it:

4.1.1 Install from source

1. cd to the repository's directory containing setup.py
2. Use your installation's Pip:
    - `pip install .` (for *Release* mode)
    - `pip install -e .[dev]` (for *Development/Editable* mode)

In *development mode*, Pip will install Pytest and some linters helpful while developing, and create symbolic links under Python's packages directory instead of copying *FIDL* to it. This allows you to modify your `.py` files and test on the fly, without need to reinstall every time you make a change.

4.1.2 Install from PyPi

FIDL is in PyPi. If you are able to reach PyPi, installing is as easy as:

```
pip install FIDL
```

4.1.3 Running tests

Load the test IDB `putty.i64` (under *tests/data*) in IDA.

Now simply execute the `pytest_fidl.py` script (under *tests*) from within IDA (`Alt + F7`)

Warning: There is an issue related to testing with Pytest in Python3. Tests are not working for IDA with Python3 at the moment.

4.2 Known gotchas

New in version 1.0.

4.2.1 Importing the module

To import _FIDL_ into your own programs, use the uppercase form of the name, that is:

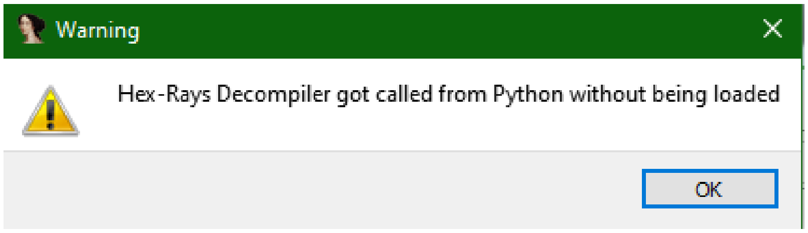
`import FIDL` or `import FIDL.decompiler_utils as du` will work, but

`import fidl` will result in an import error

4.2.2 Bitness mismatch

Getting the following error message:

```
"Hex-Rays Decompiler got called from Python without being loaded"
```



Hex-Rays Decompiler got called from Python without being loaded

usually means that you are trying to decompile an x86 binary with the x86_64 version of IDA. The binary will still load and be analyzed but the decompiler plugin will fail.

4.3 Getting started

New in version 1.0.

This section will introduce the basics of the FIDL API as well as examples for the most common tasks.

4.3.1 The ControlFlowinator

The main object for the FIDL API is a data structure representing individual functions. This data structure is a mix between the assembly-level control flow graph (*CFG*) and the decompilation output in IDA. It has been conveniently named `controlFlowinator`.

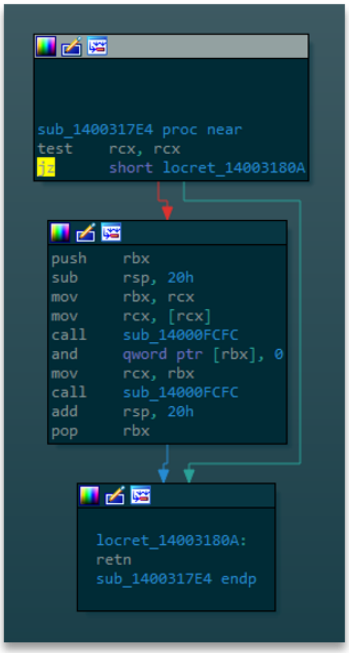
To understand it better, picture yourself a *CFG* where every node is a high level code construct, e.g.,

- if
- assignment
- function call

- return
- etc.

In a `controlFlowinator` object, every node of the *CFG* translates roughly to a line of decompiled code.

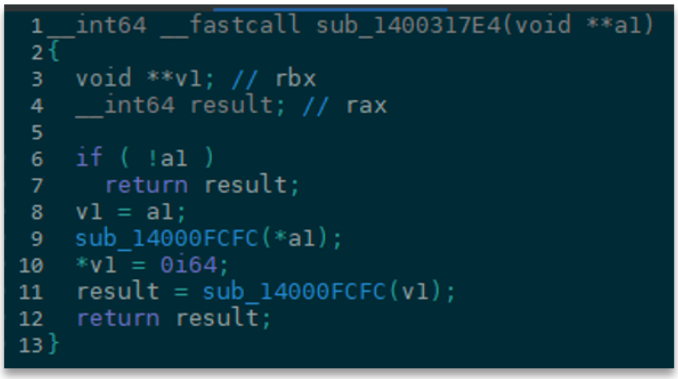
Below you can find a visualization of the `controlFlowinator` object for an example function, next to the *classical* IDA function views (assembly *CFG* and decompilation). Essentially, FIDL adds a new abstract representation of a function.



```
sub_1400317E4 proc near
test    rcx, rcx
jz      short locret_14003180A
```

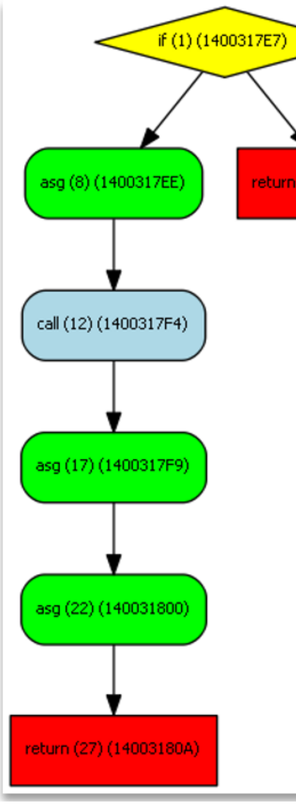
```
push    rbx
sub     rsp, 20h
mov     rbx, rcx
mov     rcx, [rcx]
call    sub_14000FCFC
and     qword ptr [rbx], 0
mov     rcx, rbx
call    sub_14000FCFC
add     rsp, 20h
pop     rbx
```

```
locret_14003180A:
retn
sub_1400317E4 endp
```

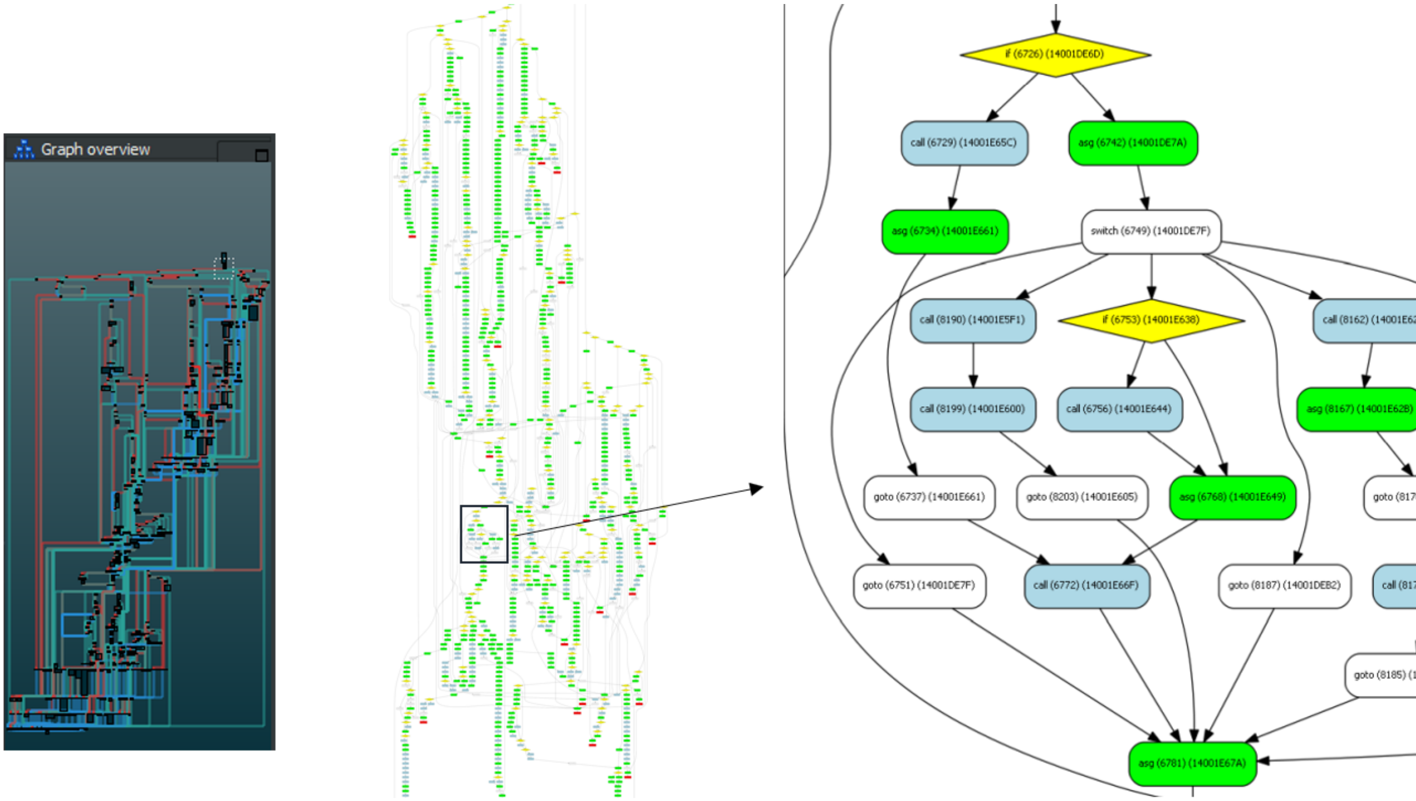


```
__int64 __fastcall sub_1400317E4(void **a1)
{
  void **v1; // rbx
  __int64 result; // rax

  if ( !a1 )
    return result;
  v1 = a1;
  sub_14000FCFC(*a1);
  *v1 = 0i64;
  result = sub_14000FCFC(v1);
  return result;
}
```



Creating a `controlFlowinator` object scales well when dealing with large functions:



4.3.2 Batteries included

The `controlFlowinator` object contains by default a lot of interesting information about the function it represents, e.g.,

- local variables
- arguments
- function calls
- return type

This information is easily accessible as attributes. Let's use the following function (from `putty.exe`) as an example:

```
BOOL __fastcall complex_75_sub_140062678(__int64 a1, const WCHAR *a2, __int64 a3, int a4)
{
  __int64 v4; // rdi
  const __m128i *v5; // rbx
  int v6; // eax
  SIZE_T v7; // r15
  _DWORD *v8; // rax
  void *v9; // r14
  HGLOBAL v10; // rax
  void *v11; // r13
  __m128i *v12; // r12
  int v13; // esi
  <snip...>
```

This is a fairly complex function with four arguments and many local variables.

Function arguments

Extract information from a function arguments is easy. We will start by importing the module and creating a `controlFlowinator` object.

```
Python>import FIDL.decompiler_utils as du
Python>c = du.controlFlowinator(ea=here(), fast=False)
Python>c
<FIDL.decompiler_utils.controlFlowinator instance at 0x00000176B566BE48>
```

We can now access this function arguments via the `args` attribute. Note that arguments are pretty printed by default.

```
Python>c.args
Name: a1
  Type name: __int64
  Size: 8
Name: a2
  Type name: const WCHAR *
  Size: 8
Complex type: WCHAR
Pointed object: const WCHAR
Name: a3
  Type name: __int64
  Size: 8
Name: a4
  Type name: int
  Size: 4
{0x0: , 0x1: , 0x2: , 0x3: }
```

`c.args` is a dict indexed by a numerical index. Its individual arguments are of type `my_var_t`. Please refer to *Core API* for more information about this class.

We can now easily extract information from individual arguments. As an example we'll query properties from the first two arguments of this function.

Remember the prototype is: `BOOL __fastcall complex_75_sub_140062678(__int64 a1, const WCHAR *a2, __int64 a3, int a4)`

```
Python>first = c.args[0]
Python>dir(first)
['__doc__', '__init__', '__module__', '__repr__', '_get_var_type', 'array_type',
↪'complex_type', 'is_a_function_of', 'is_arg', 'is_array', 'is_constrained', 'is_
↪initialized', 'is_pointer', 'is_signed', 'is_tainted', 'name', 'pointed_type', 'size
↪', 'ti', 'type_name', 'var']
Python>first.name
'a1'
Python>first.type_name
'__int64'
Python>first.pointed_type
Python>first.is_signed
True
Python>first.is_pointer
False
Python>first.is_array
False

Python>second = c.args[1]
Python>second.name
```

(continues on next page)

(continued from previous page)

```
'a2'
Python>second.is_pointer
True
Python>second.pointed_type
const WCHAR
Python>second.type_name
'const WCHAR *'
```

See *Core API* for more information about working with arguments.

Local variables

Working with a function's local variables is very similar to working with arguments (under the hood, both are of the same type in *Hex-Rays*). In *FIDL*, local variables share type with function arguments as well (`my_var_t`).

Let's start as usual by importing the module and constructing a `controlFlowinator` object:

```
Python>import FIDL.decompiler_utils as du
Python>c = du.controlFlowinator(ea=here(), fast=False)
Python>c
<FIDL.decompiler_utils.controlFlowinator instance at 0x000001D756DB21C8>
```

Accessing the local variables using the `lvars` attribute, a dictionary of `my_var_t` objects:

```
Python>c.lvars
Name: v4
  Type name: __int64
  Size: 8
Name: v5
  Type name: const __m128i *
  Size: 8
Complex type: __m128i
Pointed object: const __m128i
<snip...>
Name: WideCharStr
  Type name: __int16[256]
  Size: 512
Array type: __int16
Name: v86
  Type name: __int64
  Size: 8
Name: vars30
  Type name: int
  Size: 4
<snip...>
```

Let's inspect an interesting one. That array of "words" for example. We happen to know the index (dict key) but we could search for the name as well by iterating the *dict* and accessing the `name` attribute. This is an straightforward exercise left to the reader ;)

```
Python>lv = c.lvars[0x55]
Python>lv.is_array
True

Python>lv
```

(continues on next page)

(continued from previous page)

```
Name: WideCharStr
  Type name: __int16[256]
  Size: 512
Array type: __int16
Array element size: 2
Array length: 256

Python>lv.array_len
0x100L
```

As we can see we have easy access to all array properties (type, length, etc.)

See *Core API* for more information about working with local variables.

Function calls

Another very important piece of information is which functions are being called by the function we are currently analyzing, as well as their arguments and return types.

For this example let's analyze another function. The function shown below displays *PuTTY*'s license:

```
INT_PTR __fastcall DialogFunc(HWND a1, int a2, unsigned __int16 a3)
{
  HWND v3; // rdi
  int v4; // edx
  int v5; // edx
  CHAR *v7; // rbx

  v3 = a1;
  v4 = a2 - 16;
  if ( !v4 )
    goto LABEL_11;
  v5 = v4 - 256;
  if ( !v5 )
  {
    v7 = sub_14000F698("%s Licence", "PuTTY");
    SetWindowTextA(v3, v7);
    sub_14000FCFC(v7);
    SetDlgItemTextA(
      v3,
      1002,
      "PuTTY is copyright 1997-2017 Simon Tatham.\r\n"
      "\r\n"
      "Portions copyright Robert de Bath, Joris van Rantwijk, Delian Delchev, Andreas Schultz, Jeroen Massar, Wez Furlong"
      ", Nicolas Barry, Justin Bradford, Ben Harris, Malcolm Smith, Ahmad Khalifa, Markus Kuhn, Colin Watson, Christopher"
      " Staite, and CORE SDI S.A.\r\n"
      "\r\n"
      "Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated docum"
      "entation files (the \"Software\"), to deal in the Software without restriction, including without limitation the r"
      "ights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to per"
      "mit persons to whom the Software is furnished to do so, subject to the following conditions:\r\n"
```

(continues on next page)

(continued from previous page)

```
      "\r\n"
      "The above copyright notice and this permission notice shall be included in all␣
↪copies or substantial portions of t"
      "he Software.\r\n"
      "\r\n"
      "THE SOFTWARE IS PROVIDED \"AS IS\", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR␣
↪IMPLIED, INCLUDING BUT NOT LIMITED TO"
      " THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND␣
↪NONINFRINGEMENT.  IN NO EVENT SHALL THE C"
      "OPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER␣
↪IN AN ACTION OF CONTRACT, TORT OR OT"
      "HERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR␣
↪OTHER DEALINGS IN THE SOFTWARE.");
    return 1i64;
  }
  if ( v5 == 1 && a3 - 1 <= 1 )
LABEL_11:
    EndDialog(a1, 1i64);
  return 0i64;
}
```

To illustrate how to work with function calls, let's get the *license string*, that is, the third argument of the `SetDlgItemTextA` function.

We will start, as usual, by creating a `controlFlowinator` object and inspecting its attributes, in this case `calls`:

```
Python>import FIDL.decompiler_utils as du
Python>c = du.controlFlowinator(ea=here(), fast=False)
Python>c
<FIDL.decompiler_utils.controlFlowinator instance at 0x000002A0B67F5B08>
```

Accessing the `calls` attribute we can quickly preview the information it contains, since it is pretty printed by default:

```
Python>c.calls
--------------------------------------
Ea: 14005892E
Target's Name: sub_14000FCFC
Target's Ea: 14000FCFC
Target's ret: __int64
Args:
Name: v7
  Type name: CHAR *
  Size: 8
  Complex type: CHAR
  Pointed object: CHAR
 - 0: Rep(type='var', val=)
--------------------------------------
Ea: 140058917
Target's Name: sub_14000F698
Target's Ea: 14000F698
Target's ret: __int64
Args:
--------------------------------------
Ea: 1400588F6
Target's Name: EndDialog
Target's Ea: 140090898
Target's ret: BOOL
```

(continues on next page)

(continued from previous page)

```
Args:
Name: a1
  Type name: HWND
  Size: 8
  Complex type: HWND__
  Pointed object: HWND__
 - 0: Rep(type='var', val=)
 - 1: Rep(type='number', val=1L)
--------------------------------------
Ea: 140058925
Target's Name: SetWindowTextA
Target's Ea: 1400909A8
Target's ret: BOOL
Args:
Name: v3
  Type name: HWND
  Size: 8
  Complex type: HWND__
  Pointed object: HWND__
 - 0: Rep(type='var', val=)
Name: v7
  Type name: CHAR *
  Size: 8
  Complex type: CHAR
  Pointed object: CHAR
 - 1: Rep(type='var', val=)
--------------------------------------
Ea: 140058942
Target's Name: SetDlgItemTextA
Target's Ea: 140090948
Target's ret: BOOL
Args:
Name: v3
  Type name: HWND
  Size: 8
  Complex type: HWND__
  Pointed object: HWND__
 - 0: Rep(type='var', val=)
 - 1: Rep(type='number', val=1002L)
 - 2: Rep(type='string', val='PuTTY is copyright 1997-2017 Simon Tatham.\r\n\r\nPortions copyright Robert de Bath, Joris van Rantwijk, Delian Delchev, Andreas Schultz, <snip...>')
[, , , , ]
```

As we can see, the long string containing *PuTTY*'s license is indeed recognized as the third argument of that Windows API. Notice how the function arguments are represented by a `named tuple` with elements `type` and `val`. We'll now programatically search the function call matching that API name:

```
Python>for k in c.calls:
Python>    if k.name == 'SetDlgItemTextA':
Python>        break
Python>
Python>k
--------------------------------------
Ea: 140058942
Target's Name: SetDlgItemTextA
```

(continues on next page)

(continued from previous page)

```
Target's Ea: 140090948
Target's ret: BOOL
Args:
Name: v3
  Type name: HWND
  Size: 8
  Complex type: HWND__
  Pointed object: HWND__
 - 0: Rep(type='var', val=)
 - 1: Rep(type='number', val=1002L)
 - 2: Rep(type='string', val='PuTTY is copyright 1997-2017 Simon Tatham.
↪\r\n\r\nPortions copyright Robert de Bath, Joris van Rantwijk <snip...>')
```

Finally, let's locate its third argument and extract its value:

```
Python>k.args
{0x0: ('var', 0x3), 0x1: ('number', 0x3eaL), 0x2: ('string', 'PuTTY is copyright 1997-
↪2017 Simon Tatham.<snip...>')}
Python>lic = k.args[2]
Python>lic.type
'string'
Python>s = lic.val
Python>s
'PuTTY is copyright 1997-2017 Simon Tatham.\r\n\r\nPortions copyright Robert de Bath,␣
↪Joris van Rantwijk, Delian Delchev, Andreas Schultz, Jeroen Massar, Wez Furlong,␣
↪Nicolas Barry, Justin Bradford, Ben Harris, Malcolm Smith, Ahmad Khalifa, Markus␣
↪Kuhn, Colin Watson, Christopher Staite, and CORE SDI S.A.\r\n\r\nPermission is␣
↪hereby granted, free of charge, to any person obtaining a copy of this software and␣
↪associated documentation files (the "Software"), to deal in the Software without␣
↪restriction, including without limitation the rights to use, copy, modify, merge,␣
↪publish, distribute, sublicense, and/or sell copies of the Software, and to permit␣
↪persons to whom the Software is furnished to do so, subject to the following␣
↪conditions:\r\n\r\nThe above copyright notice and this permission notice shall be␣
↪included in all copies or substantial portions of the Software.\r\n\r\nTHE SOFTWARE␣
↪IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING␣
↪BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR␣
↪PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR␣
↪ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR␣
↪OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR␣
↪OTHER DEALINGS IN THE SOFTWARE.'
```

Note: The function arguments of a `controlFlowinator`, representing a function, and the function arguments of a specific occurrence of a function call are not of the same type.

A **function call** can have explicitly defined constants or strings as arguments, eg. `sub_140021F58("my_string", 1337, v8)` accessed via a *named tuple* as shown in the code snippet above.

The function arguments of a `controlFlowinator` instance, representing the function itself, eg. `sub_140021F58(char *a1, int a2, __int64 a3)` are of type `my_var_t`

However, if the function call has an argument of type *var*, its *val* (ue) will be an instance of `my_var_t`

A little example

No reversing automation project is complete without an example involving `GetProcAddress`. Let's consider the following *PuTTY* function, resolving dynamically some APIs.

You can find this function at address 0x140055674 within the provided `putty.i64` IDB file (under *tests*)

```
__int64 cgp_sneaky_direct_asg()
{
  HMODULE v0; // rax
  HMODULE v1; // rbx

  v0 = sub_140065B68("comctl32.dll");
  v1 = v0;
  if ( v0 )
    qword_1400C0DD0 = GetProcAddress(v0, "InitCommonControls");
  else
    qword_1400C0DD0 = 0i64;
  if ( v1 )
    qword_1400C0DD8 = GetProcAddress(v1, "MakeDragList");
  else
    qword_1400C0DD8 = 0i64;
  if ( v1 )
    qword_1400C0DE0 = GetProcAddress(v1, "LBItemFromPt");
  else
    qword_1400C0DE0 = 0i64;
  if ( v1 )
    qword_1400C0DE8 = GetProcAddress(v1, "DrawInsert");
  else
    qword_1400C0DE8 = 0i64;
  return qword_1400C0DD0();
}
```

As we can see, some functions belonging to *comctl32.dll* are being resolved at runtime and pointers to them are stored in global variables. Since we will be seeing these global variables somewhere else in the binary, it would be good to rename them in a way that references the API they are pointing to.

The following script implements this:

```
import FIDL.decompiler_utils as du


callz = du.find_all_calls_to_within(f_name='GetProcAddress', ea=here())
for co in callz:
    # The *second* argument of ``GetProcAddress`` is the API name
    api_name = co.args[1].val

    # double check :)
    if not du.is_asg(co.node):
        continue

    lhs = co.node.x
    if du.is_global_var(lhs):
        g_addr = du.value_of_global(lhs)
        new_name = "g_ptr_{}".format(api_name)
        MakeName(g_addr, new_name)
```

The script assumes that the GUI cursor is within the function we are modifying.

First we get a list of `callObj` objects representing all occurrences of a call to `GetProcAddress` (line 4). At line 7 we extract the value of their second arguments, that is, the string containing the API names. After checking that we are indeed dealing with an assignment (something of the form `global_var = call_to_func(x, y)`), we take the left hand side of the expression (line 13). If this is indeed a global variable, we rename it to match the API it is pointing to (lines 14-17).

After executing the script the function will now look like this:

```
__int64 cgp_sneaky_direct_asg()
{
  HMODULE v0; // rax
  HMODULE v1; // rbx

  v0 = sub_140065B68("comctl32.dll");
  v1 = v0;
  if ( v0 )
    g_ptr_InitCommonControls = GetProcAddress(v0, "InitCommonControls");
  else
    g_ptr_InitCommonControls = 0i64;
  if ( v1 )
    g_ptr_MakeDragList = GetProcAddress(v1, "MakeDragList");
  else
    g_ptr_MakeDragList = 0i64;
  if ( v1 )
    g_ptr_LBItemFromPt = GetProcAddress(v1, "LBItemFromPt");
  else
    g_ptr_LBItemFromPt = 0i64;
  if ( v1 )
    g_ptr_DrawInsert = GetProcAddress(v1, "DrawInsert");
  else
    g_ptr_DrawInsert = 0i64;
  return g_ptr_InitCommonControls();
}
```

You can find this script under *examples/getprocaddr_renaming_globals.py* in the source code distribution.

A more complete example

Let’s take a look at a contrived example to showcase a typical use of the FIDL API. The example has been taken from @fabs0x0 presentation about Joern (a source code static analysis tool).

The problem we are trying to solve is the following: find all the functions allocating memory using `malloc` in a way that its size can overflow, that is, of the form `len + imm`. Afterwards, find occurrences of `memcpy` where the same variable `len` is used as a *size* parameter.

The example script can be found on the **examples** directory of the source code distribution, along with the *IDB* file of a simple program implementing this potentially vulnerable code pattern. The same script is displayed below:

```
# ---------------------------------------------------------------
# Example from @fabs0x0 presentation about Joern
# https://fabs.codeminers.org/talks/2019-joern.pdf
#
# Note: this example is deliberately verbose.
# There are cleaner, leaner ways to implement this idea
# but the objective here is to  showcase the API.
# ---------------------------------------------------------------

```

(continues on next page)

(continued from previous page)

```python
from ida_hexrays import cot_add
import FIDL.decompiler_utils as du


def find_possible_malloc_issues(c=None):
    """Searches for instances where malloc argument may wrap around
    and there's a dangerous use of it in a memory write operation.

    :param c: a :class:`controlFlowinator` object
    :type c: :class:`controlFlowinator`
    :return: a list of dict containing free-form information
    :rtype: list
    """

    results = []
    suspicious_lens = []

    mallocz = du.find_all_calls_to_within('malloc', c.ea)
    memcpyz = du.find_all_calls_to_within('memcpy', c.ea)

    if not mallocz or not memcpyz:
        return []

    # Check whether the ``malloc`` call contains an arithmetic
    # expression as function argument. We are only looking
    # for additions in this case
    for co in mallocz:
        m_arg = co.args[0]
        if m_arg.type != 'unk':
            continue

        is_ari = du.is_arithmetic_expression(
            m_arg.val,
            only_these=[cot_add])

        if is_ari:
            # Now, there are many ways to skin a cat...
            # we'll use the following on this example.
            # Assuming ``len + <number>`` -> ``len``
            lhs = m_arg.val.x  # looking for var_ref_t
            rhs = m_arg.val.y  # looking for an immediate

            if du.is_var(lhs) and du.is_number(rhs):
                real_var = du.ref2var(ref=lhs, c=c)

                # This is not strictly necessary but it is
                # recommended to use ``my_var_t`` objects if
                # possible, since they contain a lot of useful
                # properties/methods
                my_var = du.my_var_t(real_var)

                suspicious_lens.append(my_var)

    # Are there any of these "suspicious" length variables
    # being used in a memcpy?
    for lv in suspicious_lens:
        for co in memcpyz:
```

(continues on next page)

(continued from previous page)

```
67             # memcpy(src, dst, size)  // size: 3rd arg
68             sv = co.args[2]
69 
70             # Checking whether the `size` parameter is a variable,
71             # it could be a constant as well...
72             if sv.type == 'var':
73                 v_name = sv.val.name
74                 # Checking whether two local variables are the same
75                 # is better done by comparing their names.
76                 if lv.name == v_name:
77                     res = {
78                             'ea': c.ea,
79                             'msg': "Check use of {} at {:X}".format(
80                                 lv.name,
81                                 co.ea,
82                                 )}
83                     results.append(res)
84 
85     return results
86 
87 
88 def main():
89     results = du.do_for_all_funcs(
90         find_possible_malloc_issues,
91         min_size=0,
92         fast=False)
93 
94     print "=" * 80
95     print results
96 
97 
98 if __name__ == '__main__':
99     main()
```

As we can see, *The ControlFlowinator* object is indeed the central piece of this API. It is the only argument of the function `find_possible_malloc_issues` at line 14. The convenience function `do_for_all_funcs` (line 89) is used to iterate over all functions on a binary, calculate their `controlFlowinator` and call a function with it as parameter (see line 90) and the API documentation for more information about this wrapper.

At lines 27, 28 all occurrences of calls to `malloc` and `memcpy` are calculated. The result of `find_all_calls_to_within` are so called `callObj`, a complex data structure containing a lot of information about the *call* (name, arguments, location, etc.)

The argument of `malloc` is used as a parameter of `is_arithmetic_expression` (line 41), an auxiliary function returning a *boolean*, indicating whether the expression is arithmetic (that is, addition, substraction, multiplication, etc. or a combination of them). In this specific case we specify a second parameter to restrict the search to additions only.

If an expression representing an addition (a + b) is found we extract their operands {a, b} (lines 49, 50). Afterwards, we check whether the operands are of the *type* we are looking for, that is, a variable and a number (line 52). If this is true, we have found one of these `len` variables of interest, so we create `my_var_t` object and save it in a list for later usage (lines 59, 61). For more information on `my_var_t` objects please refer to the *Local variables* section.

Now that we have a list of *suspicious* `len` variables in this function is time to go over all calls to `memcpy`, get their third arguments (line 68) and get their names (line 73). This is done only in the case that the *size* parameter is a variable (line 72), since it could be a constant value as well.

Finally, we compare the names of the two variables (line 76) and save the results in a JSON-like format to be returned at the end of the script's execution.

Running this over the example *IDB* provided, produces the expected result (line 6):

```
1 40118A: variable 'v17' is possibly undefined
2 <snip...>
3 401C88: positive sp value 18 has been found
4 401CBA: could not find valid save-restore pair for ebx
5 ==========================================================================
6 [{'msg': 'Check use of len at 401030', 'ea': 4198400L}]
```

4.4 Core API

New in version 1.0.

This section documents the FIDL core API, and it's intended for developers of IDA plugins

4.4.1 API Overview

class `decompiler_utils.`**`BBGraph`**(*f_ea*)

Representation of the assembly CFG for a function

`find_connected_paths`(*bb_start*, *bb_end*, *co=10*)

Leverages NetworkX to find all connected paths

Parameters

- **`bb_start`** (*`Basic block`*) – Initial basic block
- **`bb_end`** (*`Basic block`*) – Final basic block
- **`co`** (*`int, optional`*) – Cutoff parameter

NOTE: the cutoff parameter in `nx.all_simple_paths` serves two purposes:

- reduce the chances of CPU melting (algo is O(n!))
- nobody will inspect (manually) monstruous paths

Returns generator of lists or None

`get_node`(*addr*)

Given a function's address, returns the basic block (address) that contains it (or None)

Parameters **`addr`** (*`int`*) – address within a function

Returns Address of the node containing the input address

Return type int

`decompiler_utils.`**`NonLibFunctions`**(*start_ea=None*, *min_size=0*)

Generator yielding only non-lib functions

Parameters

- **`start_ea`** (*`int, optional`*) – Address to start looking for non-library functions.
- **`min_size`** (*`int, optional`*) – Minimum function size. Useful to filter small, uninteresting functions.

decompiler_utils.**all_paths_between**(*c*, *start_node=None*, *end_node=None*, *co=40*)

Calculates all paths between `start_node` and `end_node`

Calculating paths is one of these things that is better done with the paralell index graph (`c.i_cfg`) It haywires when done with complex elements.

FIXME: the co (cutoff) param is necessary to avoid complexity explosion. However, there is a problem if it's reached...

Parameters

- **c** (*controlFlowinator*) – a *controlFlowinator* object
- **start_node** (cexpr_t) – a *controlFlowinator* node
- **start_node** – a *controlFlowinator* node
- **co** (*int, optional*) – the *cutoff* value controls the maximum path length.

Returns it **yields** a list of nodes for each path

Return type list

decompiler_utils.**assigns_to_var**(*cex*)

Does this :class:`cexpr_t` assign a value to any variable?

TODO: this is limited for now to expressions of the type:

```
v1 = something something
```

Parameters **cex** (cexpr_t) – a `cexpr_t` object

Returns the assigned var index (to `cf.lvars` array) or -1 if the `cexpr_t` does not assign to any variable

Return type int

decompiler_utils.**blowup_expression**(*cex*, *final_operands=None*)

Extracts all elements of an expression

Ex: `x + 1 < y` -> `{x, 1, y}`

Parameters **cex** (cexpr_t) – a `cexpr_t` object

Returns a *set* of elements (the *final_operands*)

Return type set

class decompiler_utils.**cImporter**

Collect import information

This is mainly to work around the fact that :func:`get_func_name` does not resolve imports...

get_imports_info()

class decompiler_utils.**callObj**(*c=None*, *name=''*, *node=None*, *expr=None*)

Auxiliary object for code clarity.

It represents the occurrence of a `call` expression.

Parameters

- **name** (*string, optional*) – name of the function called
- **node** (*controlFlowinator*) – a *controlFlowinator* node containing the call expression

- **expr** (cexpr_t) – the call expression element

decompiler_utils.**citem2higher**(*citem*)

This gets the higher representation of a given :class:citem, that is, a :class:cinsn_t or :class:cexpr_t

Parameters **citem** (:class:citem) – a :class:citem object

class decompiler_utils.**controlFlowinator**(*ea=None*, *fast=True*)

This is the main object of FIDL's API.

It finds all decompiled code "blocks" and recreates a CFG based on this information.

This gives us the best of both worlds: the possibility to analyze a graph (like in disassembly mode) and the power of :class:citem based analysis.

Some analysis is performed *after* the CFG has been constructed. These are rather cost intensive, so they are turned off by default. Use fast=False to apply these and get a better CFG.

Parameters

- **ea** (*int*) – address of the function to analyze
- **fast** (*bool*) – Set to False for an object with richer information

dump_cfg(*out_dir*)

Dump the CFG for debugging purposes

This dumps a representation of the CFG in DOT format. To generate an image:

```
dot.exe -Tpng decompiled.dot -o decompiled.png
```

dump_i_cfg()

Dump interim CFG for debugging purposes

decompiler_utils.**create_comment**(*c=None*, *ea=0*, *comment="*)

Displays a comment at the line corresponding to ea

TODO: avoid creating orphan comment in case the mapping from ea to decompiled code fails

Parameters

- **c** (*controlFlowinator*) – a *controlFlowinator* object
- **ea** (*int*) – address for the comment
- **comment** (*string*) – the comment to add

decompiler_utils.**debug_blownup_expressions**(*c=None*)

Debugging helper.

Show all blown up expressions for this function.

Parameters **c** (*controlFlowinator*) – a *controlFlowinator* object

decompiler_utils.**debug_get_break_statements**(*c*)

decompiler_utils.**debug_stahp**()

Toggles DEBUG value, useful for testing

decompiler_utils.**decast**(*ins*)

Remove the cast, returning the casted element

decompiler_utils.**display_all_calls_to**(*func_name*)

Wrapping *display_line_at()* since this is the most common use of this API

Parameters **func_name** (*string*) – name of the function to search references `decompiler_utils.`**`display_line_at`**`(`*ea*, *silent=False*`)`

Displays the line of pseudocode corresponding to `ea`

This is useful to quickly answer questions like:

- "Is this function always called with its first parameter being a constant?"
- "I want to see all the error messages displayed by this function"
- etc.

Parameters

- **`ea`** (*`int`*) – address of an element contained within the line to display
- **`silent`** (*`bool`*) – flag controlling verbose output

`decompiler_utils.`**`display_node`**`(`*c=None*, *node=None*, *color=None*`)`

Displays a given node in the `pseudoviewer`

Parameters

- **`c`** (*`controlFlowinator`*) – a *`controlFlowinator`* object
- **`node`** (`cexpr_t`) – a *`controlFlowinator`* node
- **`color`** (*`int, optional`*) – color to mark the line of code corresponding to *node*

`decompiler_utils.`**`display_path`**`(`*cf=None*, *path=None*, *color=None*`)`

Shows a path's code and colors its lines.

Parameters

- **`cf`** (an `cfunc_t` object, optional) – a decompilation object
- **`path`** (*`list`*) – a list of :*`controlFlowinator`* nodes
- **`color`** (*`int, optional`*) – color to mark the lines of code corresponding to *path*

Returns a list of function lines (path nodes)

Return type list

`decompiler_utils.`**`do_for_all_funcs`**`(`*func*, *fast=True*, *start_ea=None*, *blacklist=None*, *min_size=100*, ***kwargs*`)`

This is a generic wrapper for all kinds of logic that we want to apply to all the functions in the binary.

Parameters

- **`func`** (*`function`*) – function "pointer" performing the analysis. Its only mandatory argument is a *`controlFlowinator`* object.
- **`fast`** (*`boolean, optional`*) – parameter fast for the *`controlFlowinator`* object.
- **`start_ea`** (*`int, optional`*) – Address to start looking for non-library functions.
- **`blacklist`** (*`function, optional`*) – a function determining whether to process a function. Implemented via dependency injection.

Returns A list of JSON-like messages (individual function results)

Return type list

`decompiler_utils.`**`does_constrain`**`(`*node*`)`

This tries to answer the question: "Does this `node` constrains variables in any way?"

Essentially it is looking for the occurrence of variables within known *constrainer constructs*, eg. inside an `if` condition.

TODO: many more heuristics can be included here

Parameters **node** (cinsn_t or cexpr_t) – typically a *controlFlowinator* node

Returns a set of variable indexes (to cf.lvars array)

Return type set

decompiler_utils.**dprint**(*s=''*)

This will print a debug message only if debugging is active

Parameters **s** (*str, optional*) – The debug message

decompiler_utils.**dump_lvars**(*ea=0*)

Debugging helper.

decompiler_utils.**dump_pseudocode**(*ea=0*)

Debugging helper.

decompiler_utils.**find_all_calls_to**(*f_name*)

Finds all calls to a function with the given name

Note that the string comparison is relaxed to find variants of it, that is, searching for malloc will match as well _malloc, malloc_0, etc.

Parameters **f_name** (*string*) – the function name to search for

Returns a list of *callObj*

Return type list

decompiler_utils.**find_all_calls_to_within**(*f_name*, *ea*)

Finds all calls to a function with the given name within the function containing the ea address.

Note that the string comparison is relaxed to find variants of it, that is, searching for malloc will match as well _malloc, malloc_0, etc.

Parameters

- **f_name** (*string*) – the function name to search for
- **ea** (*int*) – any address within the function that may contain the calls

Returns a list of *callObj*

Return type list

decompiler_utils.**find_elements_of_type**(*cex*, *element_type*, *elements=None*)

Recursively extracts expression elements until a cexpr_t from a specific group is found

Parameters

- **cex** (cexpr_t) – a cexpr_t object
- **element_type** (a cot_xxx value (eg. cot_add)) – the type of element we are looking for (as a cot_xxx value, see compiler_consts.py)

Returns a set of cexpr_t of the specified type

Return type set

decompiler_utils.**get_all_vars_in_node**(*cex*)

Extracts all variables involved in an expression.

Parameters **cex** (cexpr_t) – typically a *controlFlowinator* node

Returns list of var_t indexes (to cf.lvars)

Return type list

decompiler_utils.**get_cfg_for_ea**(*ea*, *dot_exe*, *out_dir*)

Debugging helper.

Uses DOT to create a .PNG graphic of the ControlFlowinator CFG and displays it.

Parameters

- **ea** (*int*) – address of the function to analyze
- **dot_exe** (*string*) – path to the DOT binary
- **out_dir** (*string*) – directory to write the .DOT file

decompiler_utils.**get_cond_from_statement**(*ins*)

Given a cinsn_t representing a control flow structure (do, while, for, etc.), it returns the corresponding cexpr_t representing the condition/argument for that code construct.

This is useful since we usually want to peek into conditional statements…

Parameters **ins** (cinsn_t) – the cinsn_t associated with a control flow structure

Returns the condition or argument within that control flow structure

Return type cexpr_t

decompiler_utils.**get_function_vars**(*c=None*, *ea=0*, *only_args=False*, *only_locals=False*)

Populates a dict of *my_var_t* for the function containing the specified ea

Parameters

- **c** (*controlFlowinator*) – a *controlFlowinator* object, optional
- **ea** (*int*) – the function address
- **only_args** (*bool, optional*) – extract only function arguments
- **only_locals** (*bool, optional*) – extract only local variables

Returns A dictionary of *my_var_t*, indexed by their index

decompiler_utils.**get_interesting_calls**(*c*, *user_defined=[]*)

Not all functions are created equal. We are interested in functions with certain names or substrings in it.

Parameters

- **c** (*controlFlowinator*) – a *controlFlowinator* object
- **user_defined** (*list, optional*) – a list of names (or substrings), if not supplied a hard-coded default list will be used.

Returns a list of *callObj*

Return type list

decompiler_utils.**get_return_type**(*cf=None*)

Hack to get the return value of a function.

Parameters **cf** (ida_hexrays.cfuncptr_t) – the result of decompile()

Returns Type information for the return value

Return type tinfo_t

decompiler_utils.**is_arithmetic_expression**(*cex*, *only_these=[]*)

Checks whether this is an arithmetic expression.

Parameters

- **cex** (cexpr_t) – expression, usually this is a *node*.
- **only_these** (a list of cot_* constants, eg. cot_add.) – a list of arithmetic expressions to look for. These are defined in ida_hexrays

Returns True or False

Return type bool

decompiler_utils.**is_array_indexing**(*ins*)

decompiler_utils.**is_asg**(*ins*)

decompiler_utils.**is_binary_truncation**(*cex*)

Looking for expressions truncating a number

These expressions are of the form v1 & 0xFFFF or alike

Parameters **cex** (*:class:cexpr_t*) – an expression

Returns True or False

Return type bool

decompiler_utils.**is_call**(*ins*)

decompiler_utils.**is_cast**(*ins*)

decompiler_utils.**is_final_expr**(*cex*)

Helper for internal functions.

A final expression will be defined as one that can not be further decomposed, eg. number, var, string, etc.

Normally, you should not need to use this.

Parameters **cex** (cexpr_t) – a cexpr_t object

Returns True or False

Return type bool

decompiler_utils.**is_global_var**(*ins*)

Tells whether ins is a global variable

TODO: enhance this heuristic

Parameters **ins** – cexpr_t or insn_t

Returns True or False

Return type bool

decompiler_utils.**is_if**(*ins*)

decompiler_utils.**is_number**(*ins*)

Convenience wrapper

decompiler_utils.**is_ptr**(*ins*)

decompiler_utils.**is_read**(*ins*)

Try to find read primitives.

Looking for things like:

```
v3 = *(_DWORD *)(v5 + 784)
```

NOTE: this will find expressions that are read && write, since they are not mutually exclusive

TODO: Rather rough, it is a first version…

Parameters **node** (cinsn_t or cexpr_t) – a *controlFlowinator* node

Returns True or False

Return type bool

decompiler_utils.**is_ref**(*ins*)

decompiler_utils.**is_string**(*ins*)

Convenience wrapper

decompiler_utils.**is_var**(*ins*)

Whether this ins corresponds to a variable

Remember that if this evaluates to True, we are dealing with an object of type var_ref_t which are pretty much useless. We may want to convert this to a lvar_t and even better to a *my_var_t* afterwards.

ref2var() is a simple wrapper to perform the conversion between reference and variable

decompiler_utils.**is_write**(*node*)

Try to find write primitives.

Looking for things like:

```
*(_DWORD *)(something) = v38
arr[i] = v21
```

TODO: Rather rough, it is a first version…

Parameters **node** (cinsn_t or cexpr_t) – a *controlFlowinator* node

Returns True or False

Return type bool

decompiler_utils.**lex_citem_indexes**(*line*)

Part of Lighthouse plugin

Lex all ctree item indexes from a given line of text. The HexRays decompiler output contains invisible text tokens that can be used to attribute spans of text to the ctree items that produced them.

decompiler_utils.**lines_and_code**(*cf=None*, *ea=0*)

Mapping of line numbers and code

Parameters

- **cf** (an cfunc_t object, optional) – a decompilation object
- **ea** (*int, optional*) – Address within the function to decompile, if no *cf* is provided

Returns a dictionary of lines of code, indexed by line number

Return type dict

decompiler_utils.**main**()

decompiler_utils.**map_citem2line**(*line2citem*)

Part of Lighthouse plugin

Creates a mapping of citem indexes to lines of code

decompiler_utils.**map_line2citem**(*decompilation_text*)

Part of Lighthouse plugin

Map decompilation line numbers to citems. This function allows us to build a relationship between citems in the ctree and specific lines in the hexrays decompilation text.

`decompiler_utils.`**`map_line2node`**(*cfunc*, *line2citem*)

Part of Lighthouse plugin

Map decompilation line numbers to node (basic blocks) addresses. This function allows us to build a relationship between graph nodes (basic blocks) and specific lines in the hexrays decompilation text.

`decompiler_utils.`**`map_node2lines`**(*line2node*)

Part of Lighthouse plugin

Creates a mapping of nodes to lines of code

`decompiler_utils.`**`my_decompile`**(*ea=None*)

This is a workaround for the cache lifecycle problem.

It calls the *`pseudoViewer`* API if the function is not in the cache, in order to plug it in.

Parameters **`ea`** (*`int`*) – Address within the function to decompile

Returns decompilation object

Return type a `cfunc_t`

`decompiler_utils.`**`my_get_func_name`**(*ea*)

Wrapper for `get_func_name` handling some corner cases.

Parameters **`ea`** (*`int`*) – Address of the function to resolve its name

`class` `decompiler_utils.`**`my_var_t`**(*var*)

This wraps the `lvar_t` nicely into a more usable data structure.

It aggregates several interesting pieces of information in one place. eg. `is_arg`, `is_constrained`, `is_initialized`, etc.

The most commonly used attributes for this class are:

- name
- type_name
- size
- is_arg
- is_pointer
- is_array
- is_signed

Parameters **`var`** (`lvar_t`) – an object representing a local variable or function argument

`decompiler_utils.`**`num_value`**(*ins*)

Returns the numerical value of `ins`

Parameters **`ins`** – `cexpr_t` or `insn_t`

`decompiler_utils.`**`points_to`**(*ins*)

`class` `decompiler_utils.`**`pseudoViewer`**

This wraps the *`pseudoViewer`* API neatly.

We need it because some things don’t work unless you previously visited (or are currently visiting) the function whose decompiled form you want to analyze. Thus, we are forced to “Hack like in the movies”

NOTE: the performance penalty is negligible

`OPEN_NEW = 1`

REUSE_IF_PSEUDOCODE = -1

USE_EXISTING = 0

close()

Closes the pseudoviewer widget

show(*ea=0*, *reuse=-1*)

Displays the pseudoviewer widget

Parameters

- **ea** (*int, optional*) – adress of the function to display
- **reuse** (*int, optional*) – how to reuse an existing pseudocode display, if any

decompiler_utils.**ref2var**(*ref*, *c=None*, *cf=None*)

Convenient wrapper to streamline the conversions between `var_ref_t` and `lvar_t`

Parameters

- **c** (*controlFlowinator*) – a *controlFlowinator* object, optional
- **cf** (a `cfunc_t` object) – a decompilation object (usually the result of `decompile`), optional
- **ref** (`var_ref_t`) – a reference to a variable in the pseudocode

Returns a `lvar_t` object

Return type `lvar_t`

decompiler_utils.**ref_to**(*ins*)

decompiler_utils.**string_value**(*ins*)

Gets the string corresponding to `ins`

Works with *C-str* and *Unicode*

Parameters **ins** – `cexpr_t` or `insn_t`

Returns string for this `ins`

Return type string

decompiler_utils.**value_of_global**(*ins*)

Returns the value of a global variable

Installation How to install FIDL and verify it

Known gotchas List of quirks or things good to know

Getting started Basic usage to get you started

Core API The core API documentation

Python Module Index

Index