École Polytechnique Fédérale de Lausanne

CythonPunk2077
Recovering Cython-compiled Python code

by Vlad Ciuleanu

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Luca Di Bartolomeo
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 7, 2024

Abstract

This work presents a potential approach in decompiling Python code which has been compiled into native executables (PE and ELF) using Cython. In the first part, the structure of the resulting executable is presented, the way its internal structures are ordered, how the code looks when analyzed with current, state-of-the-art tools, and why such approaches fall short. Cython translates most Python code into calls to libpython functions, but for some aspects, it introduces its own wrappers or methods to make the code more portable. Then, a way to ease manual reversing is presented. Using what (limited) support Ghidra's FIDB system offers in recognizing boilerplate functions; creating type databases, for libpython stubs and PyObject* types; and using extracted features to recognize helpers introduced by Cython at compile-time. This reduces to a graph matching problem, trying to match every function in the binary to the set of the helpers. Following, we present a way to adapt variants of the Relooper and Stackifier algorithms, specially crafted for using Python as a backend. Finally, the thesis focuses on decompiling the pCode generated by Ghidra to an output that resembles the original Python code, with modules, dictionaries, and, most importantly, structured control flow.

Contents

Chapter 1

Introduction

Cython is an optimizing static compiler for the Python programming language. It gives you the combined power of Python and C to let you write Python code that calls back and forth from and to C or C++ code natively at any point. The C code is generated once and then compiled with all major C/C++ compilers in CPython 2.6, 2.7 (2.4+ with Cython 0.20.x) as well as 3.5 and all later versions. All of this makes Cython ideal for wrapping external C libraries, embedding CPython into existing applications, and for fast C modules that speed up the execution of Python code. [2].

It implements multiple features and even its own superset language, to facilitate accelerating libraries such as Numpy. It targets multiple platforms, but for the scope of this paper, the focus will be CPython, the Python implementation in C, which is what is commonly installed on Linux and Windows machines.

1.1 Generate stand-alone executables with Cython

Even if Cython was designed as a bridge, to support calling fast, native code from Python, there is a specific use case that sparks interest: compiling Python code into stand-alone executables. All one has to do is to execute *cython* with *–embed* flag and then link against libpython when compiling the resulting *"cythonized"* C file[4]. However, the documentation on this topic isn't ideal, to say the least. The example given `https://github.com/cython/cython/blob/master/Demos/embed/Makefile` feels cumbersome and hard to follow. [1]

[1] `https://stackoverflow.com/questions/22507592/making-an-executable-in-cython`

1.2 Motivation

The Python to C compilation allows users to write in an easy-to-learn, high-level language and execute fast, native code on embedded systems. A router firmware, given as a target in pwn2own Tokyo 2023, with parts written in Python, compiled and linked into an ELF using Cython. When security researchers must analyze such programs, they are met with an unfriendly, messy decompilation with low data-to-noise ratio.

```c
    DAT_001078b0 = __Pyx_GetBuiltinName(DAT_00107878);
    if (DAT_001078b0 == 0) {
      iVar21 = 0x98f;
      iVar8 = 1;
      bVar7 = true;
      goto LAB_001039e2;
    }
    DAT_001078a0 = PyTuple_Pack(1,DAT_00107820);
    if ((DAT_001078a0 == 0) || (DAT_001078a8 = PyTuple_Pack(1,DAT_00107898
       ), DAT_001078a8 == 0)) {
      iVar21 = 0x991;
      iVar8 = 1;
      bVar7 = true;
      goto LAB_001039e2;
    }
    plVar13 = (long *)PyList_New(1);
    plVar6 = DAT_00107840;
    if (plVar13 == (long *)0x0) {
      iVar21 = 0x9a4;
      iVar8 = 1;
      bVar7 = true;
      goto LAB_001039e2;
    }
    lVar12 = plVar13[1];
    *DAT_00107840 = *DAT_00107840 + 1;
    if ((*(byte *)(lVar12 + 0xab) & 2) == 0) {
LAB_001034ea:
                    /* WARNING: Subroutine does not return */
      __assert_fail("PyList_Check(op)","/usr/include/python3.11/cpython/
          listobject.h",0x2d,
                    "PyList_SET_ITEM");
    }
    *(long **)plVar13[3] = plVar6;
    lVar12 = FUN_001029f3(DAT_00107840,plVar13);
    if (lVar12 == 0) {
      iVar21 = 0x9a9;
      iVar8 = 1;
```

Listing 1.1: Example of a cythonized program decompilation

Even if the decompiler of popular tools [2] can recover the C code, as far as we know, no research has been conducted on the possibility of recovering the original Python source. On top of that, there seems to be a lot of boilerplate and error checking, which is normally abstracted away from the user by the CPython runtime. So a new way of reversing such programs is needed.

[2] `https://dogbolt.org/?id=430d5ad0-5d9d-4b09-9290-0b52cb2e3726`

Chapter 2

Background

In order to understand how to reconstruct Python code from the cythonized C, one must first understand how the C code is generated in the first place. Start with writing a simple script, *from_import.py*:

```python
from datetime import datetime
from sys import exit

current_datetime = datetime.now()
formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted datetime:", formatted_datetime)

exit(0)
```

Listing 2.1: Sample Python code

2.1 The Cython source file

The file starts with a lot of preprocessor directives, such as *#ifdef* and *#define*, indicating support for various CPython versions, Cython versions, and platforms. The file comprises 5000 lines of code (LoC), and looking for traces of the original source code is like searching for a needle in a haystack. To locate the entry point of the program, given that the code is written in the C programming language, it is advisable to initiate the search by identifying the main function.

```c
#if PY_MAJOR_VERSION < 3
int main(int argc, char** argv)
#elif defined(_WIN32) || defined(WIN32) || defined(MS_WINDOWS)
int wmain(int argc, wchar_t **argv)
#else
static int __Pyx_main(int argc, wchar_t **argv)
#endif
{
    #if PY_MAJOR_VERSION < 3
    if (PyImport_AppendInittab("from_import", initfrom_import) < 0) return
         1;
    #else
    if (PyImport_AppendInittab("from_import", PyInit_from_import) < 0)
        return 1;
    #endif
        PyStatus status;
        PyConfig config;
        PyConfig_InitPythonConfig(&config);
        config.parse_argv = 0;
        if (argc && argv) {
            status = PyConfig_SetString(&config, &config.program_name,
                argv[0]);
            if (PyStatus_Exception(status)) {
                PyConfig_Clear(&config);
                return 1;
            }
            status = PyConfig_SetArgv(&config, argc, argv);
            if (PyStatus_Exception(status)) { ... }
        }
        status = Py_InitializeFromConfig(&config);
        if (PyStatus_Exception(status)) { ... }
        PyConfig_Clear(&config);
      PyObject* m = NULL;
      __pyx_module_is_main_from_import = 1;
      m = PyImport_ImportModule("from_import");
      if (!m && PyErr_Occurred()) {
          PyErr_Print();
          ...
      }
    }
    if (Py_FinalizeEx() < 0)
        return 2;
    return 0;
}
```

The function is pretty self-explanatory, as it does typical stuff one would expect a "main" function to do. But there is still no clear sign of the original code. The only reference is the module name, and it is a big clue because it is used as an argument to *PyImport_ImportModule*. It turns out a module is created containing the original code, and all Cython does is to call the Python interpreter to import it.

Before a module can be imported, it has to be initialized. In the demo program, this is accomplished by the call to *PyImport_AppendInittab("from_import", PyInit_from_import)*. *PyInit_from_import* is actually a function pointer: disassembling it reveals a call to *PyModuleDef_Init(&__pyx_moduledef)*. The Python documentation explains that this function "ensures a module definition is a properly initialized Python object that correctly reports its type and reference count."

Analysing the structure passed as a parameter, it seems that it has everything needed to locate the code. The documentation specifies "single-phase initialization", in which case the *PyModule_Create* function is called, and "multi-phase initialization", in which the *slots* field is populated with an array of function pointers. In the case of the demo program, it looks like this:

```
static PyModuleDef_Slot __pyx_moduledef_slots[] = {
  {Py_mod_create, (void*)__pyx_pymod_create},
  {Py_mod_exec, (void*)__pyx_pymod_exec_from_import},
  {0, NULL} };
```

The sample's code can be found in the *__pyx_pymod_exec_from_import* function.

2.1.1 Cythonized code

This function is quite large and it does not immediately begin with the code. There is a lot of boilerplate that checks the environment and initializes subcomponents:

```
  if (__Pyx_InitCachedBuiltins() < 0) __PYX_ERR(0, 1, __pyx_L1_error)
  if (__Pyx_InitCachedConstants() < 0) __PYX_ERR(0, 1, __pyx_L1_error)
    ...
  (void)__Pyx_modinit_variable_import_code();
  (void)__Pyx_modinit_function_import_code();
    ...
```

Thankfully, the code is commented, and the translation is done line by line:

```
  /* "from_import.py":4
 * from sys import exit
 *
 * current_datetime = datetime.now()             # <<<<<<<<<<<<<<
 * formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
 * print("Formatted datetime:", formatted_datetime)
 */
  __Pyx_GetModuleGlobalName(__pyx_t_2, __pyx_n_s_datetime); if (unlikely(!
      __pyx_t_2)) __PYX_ERR(0, 4, __pyx_L1_error)
  __Pyx_GOTREF(__pyx_t_2);
  __pyx_t_3 = __Pyx_PyObject_GetAttrStr(__pyx_t_2, __pyx_n_s_now); if (
      unlikely(!__pyx_t_3)) __PYX_ERR(0, 4, __pyx_L1_error)
  __Pyx_GOTREF(__pyx_t_3);
```

```
__Pyx_DECREF(__pyx_t_2); __pyx_t_2 = 0;
__pyx_t_2 = __Pyx_PyObject_CallNoArg(__pyx_t_3); if (unlikely(!__pyx_t_2
    )) __PYX_ERR(0, 4, __pyx_L1_error)
__Pyx_GOTREF(__pyx_t_2);
__Pyx_DECREF(__pyx_t_3); __pyx_t_3 = 0;
if (PyDict_SetItem(__pyx_d, __pyx_n_s_current_datetime, __pyx_t_2) < 0)
    __PYX_ERR(0, 4, __pyx_L1_error)
__Pyx_DECREF(__pyx_t_2); __pyx_t_2 = 0;
```

A few interesting things can be observed here: First, there is an error check after every function call. This is so, in case of error, a traceback is printed and the program doesn't just crash. In-between there are calls to the memory manager. Usually, these things are abstracted away from the programmer.

Sometimes CPython functions are called i.e. *PyDict_SetItem*, sometimes cython introduces wrappers like *__Pyx_GetModuleGlobalName*, which is a macro pointing to *__Pyx_GetBuiltinName*, which calls *__Pyx_PyObject_GetAttrStrNoError* finally leading to the CPython import *PyObject_GetOptionalAttr*.

__pyx_d is a dictionary containing the namespace of this module. This contains all the defined names, such as variables, functions, and classes. The assignment from the original code is implemented as to a call to *PyDict_SetItem*.

Cython introduces temporary variables. They are of the form *__pyx_t_num*. Strings are not accessed directly as literals, but as internal objects. *__pyx_n_s_current_datetime* holds the "current_datetime" string.

Consulting the Python documentation again, it seems that multiple Python objects possess namespaces implemented in dictionaries.

- For a module, global variables, defined functions, classes, and imported objects are stored here, having their names as the key.
- For a function, this dictionary contains local variables
- For a class, the field names are stored in this dictionary

Whenever a defined variable is used in Python, there is actually a lookup in a dictionary going on behind the scenes. To compile *current_datetime = datetime.now()*, Cython must retrieve the module associated with the *"datetime"* string, get the attribute identified by the *"now"* string, call this PyObject as a function, and then create a new item or assign the already existing one from the module's namespace dictionary with the key *"current_datetime"* the call's result.

2.2 Strings

Python represents strings as PyObjects. These objects are initialized at runtime from ASCII or Unicode literals.

```
PyObject *__pyx_kp_s_Formatted_datetime;
PyObject *__pyx_kp_s_Y_m_d_H_M_S;
  ...
PyObject *__pyx_n_s_current_datetime;
PyObject *__pyx_n_s_datetime;
PyObject *__pyx_n_s_exit;
  ...
PyObject *__pyx_n_s_main;
PyObject *__pyx_n_s_name;
PyObject *__pyx_n_s_now;
PyObject *__pyx_n_s_print;
  ...
PyObject *__pyx_int_0;
PyObject *__pyx_tuple__2;
PyObject *__pyx_tuple__3;
```

Objects starting with *"__pyx_kp"* represent a string literal, any other strings, such as variable and field names, start with *"__pyx_n"*.

Following the references gives this function:

```
static int __Pyx_CreateStringTabAndInitStrings(void) {
  __Pyx_StringTabEntry __pyx_string_tab[] = {
    {&__pyx_kp_u_, __pyx_k_, sizeof(__pyx_k_), 0, 1, 0, 0},
    {&__pyx_kp_s_Formatted_datetime, __pyx_k_Formatted_datetime, sizeof(
        __pyx_k_Formatted_datetime), 0, 0, 1, 0},
    {&__pyx_kp_s_Y_m_d_H_M_S, __pyx_k_Y_m_d_H_M_S, sizeof(
        __pyx_k_Y_m_d_H_M_S), 0, 0, 1, 0},
    {&__pyx_n_s__4, __pyx_k__4, sizeof(__pyx_k__4), 0, 0, 1, 1},
    {&__pyx_n_s_current_datetime, __pyx_k_current_datetime, sizeof(
        __pyx_k_current_datetime), 0, 0, 1, 1},
    ...
```

in which PyObject strings are populated from string constants within the program.

Chapter 3

Design

Armed with knowledge of Cython internals, rebuilding the original program may begin.

3.1 Identifying the strings

The *__Pyx_StringTabEntry* structure is defined as:

```c
typedef struct {
    PyObject **p;
    const char *s;
    const Py_ssize_t n;
    ... } __Pyx_StringTabEntry;
```

This could either be stored on the stack or directly in the program's sections.

In this report, we propose an approach that for each C string literal, checks all the cross-references, and if one of them has a valid pointer above and its length in a field below, then mark the pointer as a location to a PyString.

If such values are defined on the stack, Ghidra's decompiler is called to expose the stack access. Then, the stack is emulated at the *word* level, and the algorithm proceeds as in the situation above.

It is important to avoid decompiling every function, as that implies an excessively expensive computation and memory cost. Therefore, only functions referencing a C string are processed.

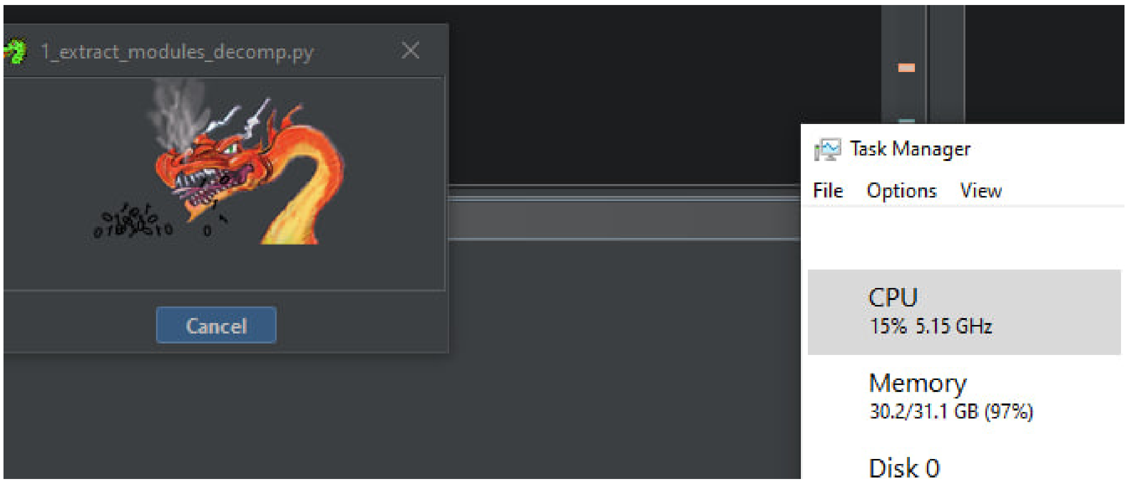


Figure 3.1: When running the tool, the pCode resulting from the decompilation of every function in the program fills out the computer's memory faster than the Java GC can free it.

3.2 Identifying the module's initialization code

When automating the steps described in the earlier sections, constructing the function call graph serves as a pivotal step. Subsequently, identify each function invoking *PyModuleDef_Init*, a crucial pursuit to pinpoint the *PyInit* function associated with the module, along with the structure housing the "slots" fields. Here, the "exec" function can be found.

3.3 Data-flow analysis

Imports, variable accesses and assignments, and attributes retrieval are done via *PyDict_GetItem*, *PyDict_SetItem*, *__Pyx_PyObject_GetAttrStrNoError* or their wrappers. Function calls are implemented in a couple of different ways, depending on the type and number of parameters. Therefore, to rebuild code, either build use-def chains for each "important" function call or backward slice the assignments.

The chosen approach is to start from an use, because some function calls do not return anything, but they are still needed for the reconstructed program to be equivalent to the original one. Take, for example, the simplest *print("Hello world!")*. It produces no data, but any decompiler must insert this call in its output.

3.4 Function identification

In order to perform data-flow analysis, one must first know where the important functions lie in the program. In the wild, most programs are stripped of any debug information or symbols, and the imports from CPython won't have their type definitions, confusing disassemblers. The Cython helpers appear as nameless, so in order for any grammar rules to work they need their names restored.

3.4.1 FIDB

FIDB, or Function Identification DataBase is Ghidra's system for library function identification, based on hashing. A program compiled with debug symbols is loaded into Ghidra, hashes are generated for every function and the results are exported to a file. On subsequent runs, when this file is loaded, Ghidra will hash all the functions from the newly loaded program and compare them to the ones present in the database.

The main downside is that for every build type ("Debug", "Release", "MinSizeRel"), platform, word size, compiler version, and the enabled optimizations the functions will be different. This implies, the FIDB approach needs an impractically large set of hashed functions.

3.4.2 Matching based on features

Every function of interest has certain features. I.e. it calls certain functions from CPython, it references a string. Therefore it is possible for every function in the program to try to match it to a certain boilerplate function added by Cython.

Analyzing, for example, the *__Pyx_PyObject_GetAttrStrNoError* wrapper, we find a couple of calls to libpython or Cython helper functions. *__Pyx_PyObject_GetAttrStr_ClearAttributeError* is called in every variant, while only one of *PyObject_GetOptionalAttr* and *_PyObject_GenericGetAttrWithDict* is called, depending on the Python version.

```
static CYTHON_INLINE PyObject* __Pyx_PyObject_GetAttrStrNoError(PyObject*
    obj, PyObject* attr_name) {
    PyObject *result;
#if __PYX_LIMITED_VERSION_HEX >= 0x030d00A1
    (void) PyObject_GetOptionalAttr(obj, attr_name, &result);
    return result;
#else
#if CYTHON_COMPILING_IN_CPYTHON && CYTHON_USE_TYPE_SLOTS && PY_VERSION_HEX
     >= 0x030700B1
    PyTypeObject* tp = Py_TYPE(obj);
    if (likely(tp->tp_getattro == PyObject_GenericGetAttr)) {
        return _PyObject_GenericGetAttrWithDict(obj, attr_name, NULL, 1);
    }
#endif
    result = __Pyx_PyObject_GetAttrStr(obj, attr_name);
    if (unlikely(!result)) {
        __Pyx_PyObject_GetAttrStr_ClearAttributeError();
    }
    return result;
```

Knowing in advance the various variants we extract the following features:

- *calls*, functions that are called in every version
- *call groups*, groups of function found only in specific versions
- *avoid*, functions which are never called by this function
- *strings*

For the Cython wrapper above the following features were extracted:

```
{
  "calls": ["__Pyx_PyObject_GetAttrStr","
      __Pyx_PyObject_GetAttrStr_ClearAttributeError"],
  "call_groups":
  [
    ["PyObject_GetOptionalAttr"],
    ["_PyObject_GenericGetAttrWithDict"]
  ]
},
```

For every function within the executable, we try to match it to a list of known helpers, using their features. It might happen that multiple functions call the the same imports, or reference the same strings. Therefore this problem resembles a bipartite graph matching, but it's a bit trickier than that. We've chosen a heuristic in order to avoid complications and incorrect matching.

3.5 Control flow rebuilding

Python is a high-level programming language, featuring structured control flow. Its most notable features are if/else clauses, while/for loops, try/catch/finally blocks, and nested function definitions. It does not support goto or any form of indirect jump out of the box [1], and switch/case support has only been introduced in version 3.10 [2].

Therefore a challenge arises when trying to generate Python code from assembly or even C, as they employ indirect jumps/calls of every kind, even allowing for non-reducible control flow graphs.

Previous work in the area [9] [5] has shown that it is possible to create a program that takes in lower-level code and produces a program in a high-level language, using structured control flow.

3.5.1 Relooper

The Emscripten's Relooper is a greedy algorithm, which tries to match structures commonly generated by *if/else* statements and *while* loops. If that is not possible, the algorithm falls to the "naive approach"[5]:

As described by the Emscripten's authors [9], a switch-in-a-loop construction is used in order to let the flow of execution move between basic blocks of code in an arbitrary manner: We set *label* to the (numerical representation of the) label of the basic block we want to reach, and do a break, which leads to the proper basic block being reached.

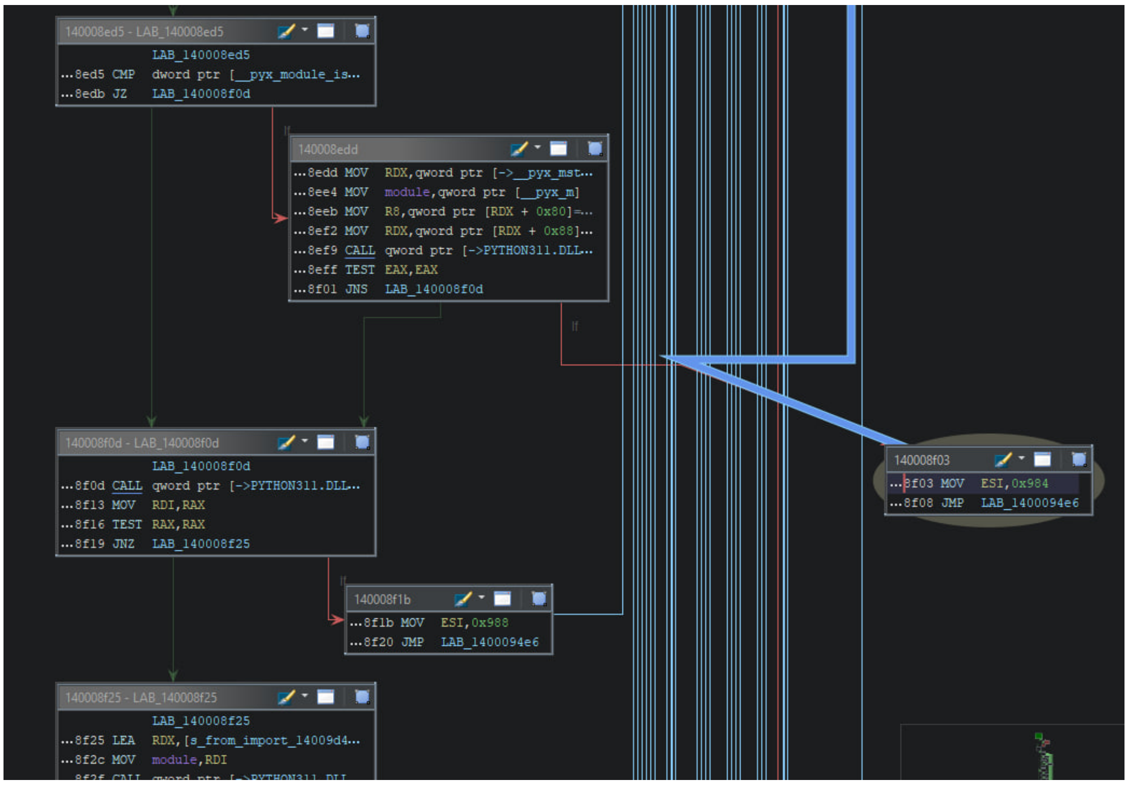
```
var label=0;
switch(A()){
  case 0:
  {
    B();
    label=3;
    break;
  }
  case 1:
  {
    label=3;
    break;
  }
  case 2:
  {
    label=4;
    break;
  }
}
```

[1] https://pypi.org/project/goto-statement/
[2] https://docs.python.org/3.10/whatsnew/3.10.html#pep-634-structural-pattern-matching

```
if(label===3){
   C();
   label=4;
}
if(label===4){
   D();
}
E();
```

Listing 3.1: Example of Javascript generated by Relooper from LLVM [5]

Usually, such a situation only happens when the control flow graph is *non-reducible*, i.e. contains loops with multiple entry points. In practice, however, a more unusual control flow can give Relooper a hard time:



```
140008ed5 - LAB_140008ed5
                LAB_140008ed5
...8ed5 CMP     dword ptr [__pyx_module_is...
...8edb JZ      LAB_140008f0d

140008edd
...8edd MOV     RDX,qword ptr [->__pyx_mst...
...8ee4 MOV     module,qword ptr [__pyx_m]
...8eeb MOV     R8,qword ptr [RDX + 0x80]=...
...8ef2 MOV     RDX,qword ptr [RDX + 0x88]...
...8ef9 CALL    qword ptr [->PYTHON311.DLL...
...8eff TEST    EAX,EAX
...8f01 JNS     LAB_140008f0d

140008f03
...8f03 MOV     ESI,0x984
...8f08 JMP     LAB_1400094e6

140008f0d - LAB_140008f0d
                LAB_140008f0d
...8f0d CALL    qword ptr [->PYTHON311.DLL...
...8f13 MOV     RDI,RAX
...8f16 TEST    RAX,RAX
...8f19 JNZ     LAB_140008f25

140008f1b
...8f1b MOV     ESI,0x988
...8f20 JMP     LAB_1400094e6

140008f25 - LAB_140008f25
                LAB_140008f25
...8f25 LEA     RDX,[s_from_import_14009d4...
...8f2c MOV     module,RDI
```

Here, the node *LAB_140008f0d* looks like it's in the true branch of *LAB_140008ed5* and in the false branch of *LAB_140008edd* (condition is inverted). The true branch of *LAB_140008edd* is an unconditional jump, going outside the scope.

3.5.2 Stackifier

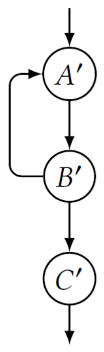
Stackifier is based upon previous work aimed at eliminating *goto*s [7]. As described in its blog post [5], its scope is to convert any reducible control flow to a structured form consisting of loops, conditional branches, and multilevel continue and break statements. For this, it employs topological sorting of the basic blocks and the use of a dominator tree. Loops are treated as infinite, with every back edge becoming a *continue* statement, and for other structures block scopes are introduced. After the code is generated, a few more passes are applied to if/else forms to make the code more readable.

Another improvement over Relooper is the way irreducible CFGs are handled. A strongly connected component algorithm is used to find all the loops. Then, a loop header is identified as a block with inbound edges from outside the strongly connected component. If more than 1 header is found, the loop code is either duplicated or the same dispatcher-based approach that Relooper uses can be implemented.

Python, however, does not have block scopes, so yet another approach is needed.

3.5.3 Glasgow Haskell Compiler

Glasgow Haskell Compiler manages to achieve a result similar to Stackifier using fewer passes, without block forms, but sacrificing code readability: The translator does move actions into if forms in a way that produces readable code, without requiring the additional steps described in Relooper. But it moves actions into loop forms too aggressively: code that we'd like to see immediately after a loop is instead placed inside the loop and followed by a br or return instruction[6].



(a) Control-flow graph

```
A:  do A'_actions
B:  do B'_actions
    if (B'_test) goto A;
C:  do C'_actions
```

(b) Equivalent Cmm

```
loop
  do A'_actions
  do B'_actions
  push B'_test
  if
    br 1  ; continue loop
  else
    do C'_actions
    return  ; exit loop
  end
end
```

(c) Translated WebAssembly

Figure 3.2: Example from [6]: Compiler aggressively includes the entire CFG in the loop

Using the strongly connected components idea from [5] we can figure out when a loop ends and can significantly improve readability.

Once such an algorithm has finished and the scopes are retrieved, use-def chains from the data-flow analysis can be translated to successfully rebuild the code.

Chapter 4

Implementation

As the disassembler we have chosen Ghidra [1], an open-source reverse engineering framework developed by the NSA. In order to interact with Ghidra's API one must either use Java, or Python via the Jython implementation. We decided to go with Python, as it's more flexible and easier to work with. However, Ghidra comes with support for Python 2.7, and, in order to support modern packages, a bridge is needed to interface with a more modern Python version. We've chosen Ghidrathon [2]

Since processing binary files and decompiling code is computationally expensive, we decided to implement caching using a sqlite database. To access this database from within Python the *sqlite* and *sqlalchemy* modules are used.

4.1 Module definition structures

In theory, one can just import the *Python.h* header file in Ghidra and all types will be available. However, the latest Ghidra version at the time of writing, 11.0.3, hangs when given such an operation. We resorted to compiling a demo program with debug symbols and creating a new *Ghidra Data Type (GDT)* archive with the types contained there.

[1] `https://ghidra-sre.org/`

[2] `https://github.com/mandiant/Ghidrathon`

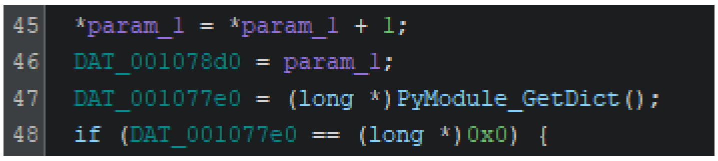
4.2 Function identification

4.2.1 Functions from libpython

Here we need to correctly set the libpython function signatures, otherwise, Ghidra fails to correctly identify the parameters passed to a function. For example:

```
PyObject *PyModule_GetDict(PyObject *module)
//Return value: Borrowed reference. Part of the Stable ABI.
```

is incorrectly recognized as having no parameter:



```
*param_1 = *param_1 + 1;
DAT_001078d0 = param_1;
DAT_001077e0 = (long *)PyModule_GetDict();
if (DAT_001077e0 == (long *)0x0) {
```

To fix this, we compiled CPython for 32 and 64 bits and generated GDTs containing all the prototypes. At run-time, we set each external symbol imported from libpython with the corresponding type, querying its function name.

4.2.2 Functions added by Cython at compile time

Identifying these is important because cythonized code usually does not call libpython directly for certain actions. If unidentified functions are left in the code we cannot match any grammar rules to translate the code.

Since these functions use macros to support a variety of platforms, architectures, and compiler versions, identifying them via FIDB or a similar hashing mechanism is not reliable in practice.

Another solution is the matching approach. We try to match every wrapper in Cython to each function in the executable, using their features. Then, a score is assigned for every pair, depending on the number of features matched. One problem that arises, is when the features of one helper is a subset of the features of another one. Depending on the order they are found in the program, they might get swapped around.

Take, for example, *__Pyx_PyObject_CallMeth0* and *__Pyx_PyObject_Call*:

```
"__Pyx_PyObject_Call":
{
  "calls": ["PyErr_Occurred", "PyErr_SetString", "PyObject_Call"],
  "call_groups":
  [
    ["Py_EnterRecursiveCall", "Py_LeaveRecursiveCall"],
    ["PyThreadState_Get", "_Py_CheckRecursiveCall"]
  ]
},
"__Pyx_PyObject_CallMeth0":
{
  "calls": ["PyErr_Occurred", "PyErr_SetString"],
  "avoid": ["PyObject_Call"],
  "call_groups":
  [
    ["Py_EnterRecursiveCall", "Py_LeaveRecursiveCall"],
    ["PyThreadState_Get", "_Py_CheckRecursiveCall"]
  ]
},
```

This is why for matching *__Pyx_PyObject_CallMeth0* we must avoid any functions calling *PyObject_Call*, as that would steal candidates from *__Pyx_PyObject_Call* by getting the same matching score.

Another idea is to divide the search in groups, depending on the number of calls present in each function, or to remove some features from the helper with more features.

4.2.3 Reducing noise

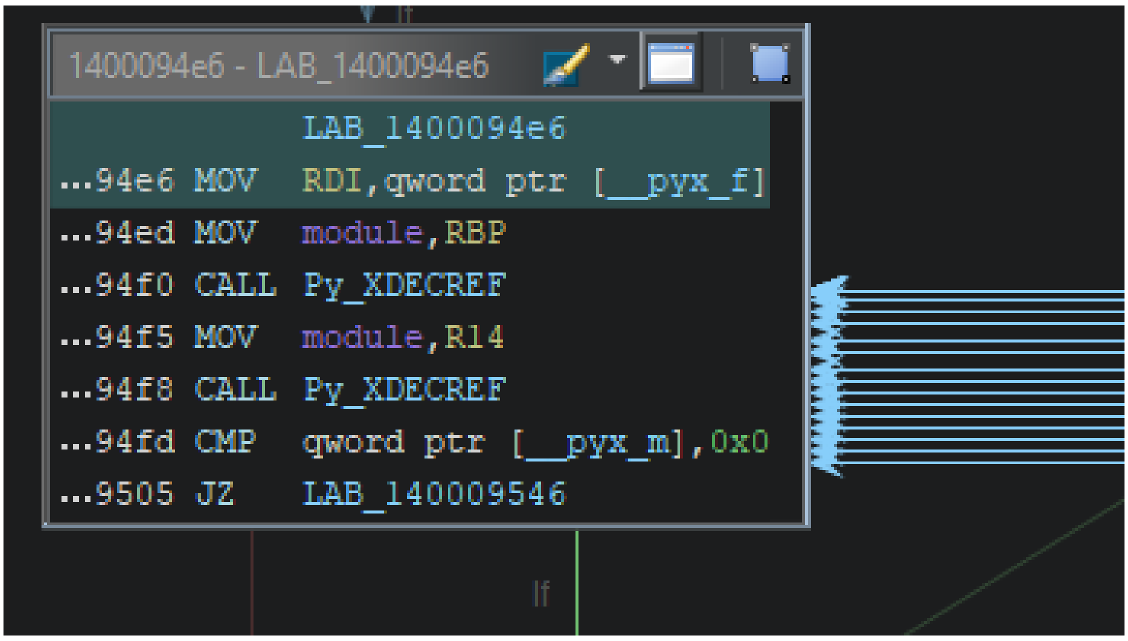
The first step in reducing noise is to kill every single block that only does *goto error*.

```
  p_Var12 = __Pyx_PyObject_GetAttrStr(p_Var10,p_Var11->
      __pyx_n_s_strftime);
  if (p_Var12 == (_object *)0x0) {
    iVar9 = 5;
    iVar4 = 0x9e1;
    goto LAB_1400094e6;
  }
```

We can identify "error nodes" by their unusually high inbound edges count:

Then, each block guarded by an if, having its only target an error node can be removed:

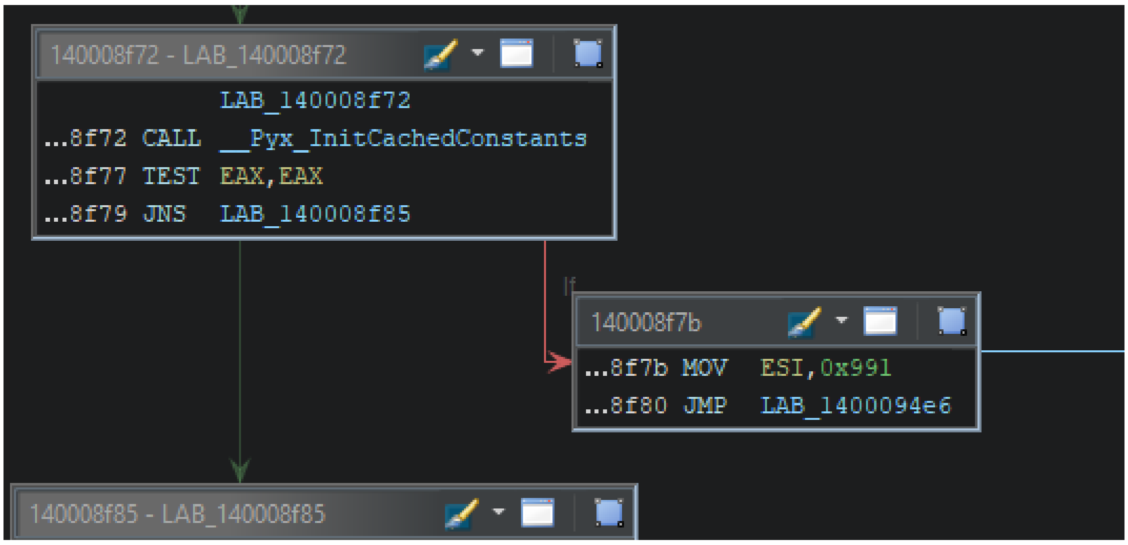
The conditional block corresponding to the *if* statement can be converted to a simple block, removing the conditional branch instruction. Unfortunately, using Ghidra, even if we manipulate



```
1400094e6 - LAB_1400094e6
                LAB_1400094e6
...94e6 MOV   RDI,qword ptr [__pyx_f]
...94ed MOV   module,RBP
...94f0 CALL  Py_XDECREF
...94f5 MOV   module,R14
...94f8 CALL  Py_XDECREF
...94fd CMP   qword ptr [__pyx_m],0x0
...9505 JZ    LAB_140009546
```

Figure 4.1: A block where errors are handled.

the pCode directly to replace the branch, the block is still recognized as a conditional.


```
140008f72 - LAB_140008f72
                 LAB_140008f72
...8f72 CALL  __Pyx_InitCachedConstants
...8f77 TEST  EAX,EAX
...8f79 JNS   LAB_140008f85
```

```
140008f7b
...8f7b MOV   ESI,0x991
...8f80 JMP   LAB_1400094e6
```

```
140008f85 - LAB_140008f85
```

Figure 4.2: A typical control-flow example implementing a "goto label" after a check.

4.2.4 Control flow rebuilding

In an ideal world, we'd scan each basic block, look for use-def chains to translate, and generate corresponding Python code. Then create a control flow graph using only the blocks that have at least one Python instruction translated. In practice, manipulating pCode basic blocks in Ghidra is not easy to do, and due to time constraints building wrapper classes over the Ghidra API was not an option. Therefore, we had to feed the entire control flow graph to the decompilation algorithm, with the only modification being tracking error nodes, direct gotos, and if clauses which check libpython functions' results for errors. Each block is translated as it is processed by Stackifier.

4.2.5 Instruction translation

Most of the time, a Python object is retrieved from a libpython function call. Then it could be either stored in RAM or directly passed around in registers to another function that processes it in some kind.

In the case of storing PyObjects to RAM, we constructed several lists of known addresses, so we could later apply copy-propagation over memory locations.

Each function call is translated separately, in a recursive descent manner, depending on which underlying operation it performs. For example, importing a module:

```
    if op.getOpcode() == PcodeOp.CALL:
            called_func_name = resolveCallName(currentProgram, op)
            if called_func_name in ["__PyxImport", "PyImport_AddModule"...
                self.imported_module_handles.append(op.getOutput())
            ...
            elif called_func_name in ["__imp_PyObject_SetAttrString", "
                PyObject_SetAttrString", "__imp_PyObject_SetAttr", "
                PyObject_SetAttr"]:
                # translates to obj.fieldname = val
                obj = reverseCopyProp(op.getInput(1), self.taint)
                fieldname = reverseCopyProp(op.getInput(2), self.taint)
                val = reverseCopyProp(op.getInput(3), self.taint)

                if val in self.imported_module_handles:
                    # we have an import, treat it differently
                    import_func = val.getDef().getInput(1)
                    imported_module_name = self.varnodeFormat(import_func)
                    print(f"{current_scope_level * 2 * ' '}import {
                        imported_module_name} as {self.varnodeFormat(
                        fieldname)}")
```

Chapter 5

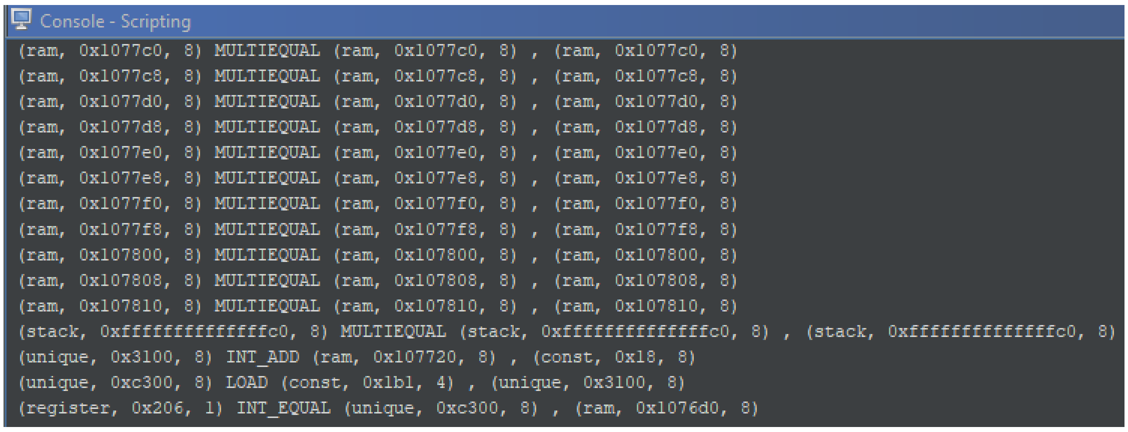
Evaluation

5.1 String identification

Executable Name	**OS/Platform**	**CPU**	**32/64 bits**	**Opt**	**Strings**	**Identified strings**
CharxWebsite	Linux GCC	armv7	32-bit	-O3	-	162
from_import	Windows VS	x86	64-bit	-O0	17	21
from_import	Windows MinGW	x86	64-bit	-O0	17	17
from_import	Windows VS	x86	64-bit	-O2	17	0 (program crash)
if_else	Android GCC	armv8	64-bit	-O0	15	15
if_else	Linux GCC	x86	64-bit	-O0	15	15
if_else	Linux GCC	x86	64-bit	-O3	15	0 (program crash)
listdir	Linux GCC	x86	64-bit	-O0	22	22

Table 5.1: Number of PyStrings recovered from each executable

The program recovers strings well. The problem lies in the data-flow analysis and is caused by the phi-nodes in the Ghidra pCode.



```
(ram, 0x1077c0, 8) MULTIEQUAL (ram, 0x1077c0, 8) , (ram, 0x1077c0, 8)
(ram, 0x1077c8, 8) MULTIEQUAL (ram, 0x1077c8, 8) , (ram, 0x1077c8, 8)
(ram, 0x1077d0, 8) MULTIEQUAL (ram, 0x1077d0, 8) , (ram, 0x1077d0, 8)
(ram, 0x1077d8, 8) MULTIEQUAL (ram, 0x1077d8, 8) , (ram, 0x1077d8, 8)
(ram, 0x1077e0, 8) MULTIEQUAL (ram, 0x1077e0, 8) , (ram, 0x1077e0, 8)
(ram, 0x1077e8, 8) MULTIEQUAL (ram, 0x1077e8, 8) , (ram, 0x1077e8, 8)
(ram, 0x1077f0, 8) MULTIEQUAL (ram, 0x1077f0, 8) , (ram, 0x1077f0, 8)
(ram, 0x1077f8, 8) MULTIEQUAL (ram, 0x1077f8, 8) , (ram, 0x1077f8, 8)
(ram, 0x107800, 8) MULTIEQUAL (ram, 0x107800, 8) , (ram, 0x107800, 8)
(ram, 0x107808, 8) MULTIEQUAL (ram, 0x107808, 8) , (ram, 0x107808, 8)
(ram, 0x107810, 8) MULTIEQUAL (ram, 0x107810, 8) , (ram, 0x107810, 8)
(stack, 0xffffffffffffffc0, 8) MULTIEQUAL (stack, 0xffffffffffffffc0, 8) , (stack, 0xffffffffffffffc0, 8)
(unique, 0x3100, 8) INT_ADD (ram, 0x107720, 8) , (const, 0x18, 8)
(unique, 0xc300, 8) LOAD (const, 0x1b1, 4) , (unique, 0x3100, 8)
(register, 0x206, 1) INT_EQUAL (unique, 0xc300, 8) , (ram, 0x1076d0, 8)
```

Figure 5.1: Phi-node instructions cluttering memory addresses exploding the search space for a def-use chain

5.2 Function arguments

In Python, positional arguments are stored in tuples, and named ones in dictionaries. Unfortunately, most of time the tool fails to correctly identify and rebuild tuples.

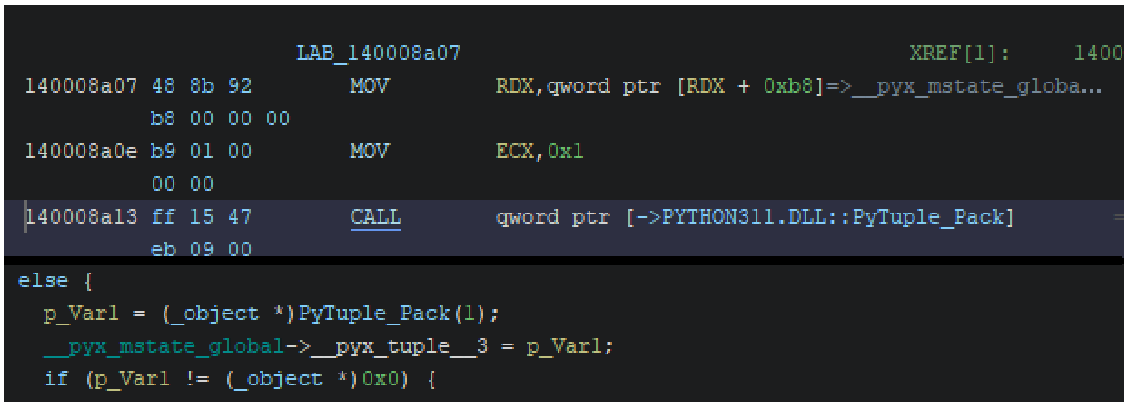
```
current_datetime.strftime(args=(register, 0x0, 8) = __imp_PyTuple_Pack (a)
    , kw=0x0i64)
random.randint(args=(0, None), kw=0x0i64)
(register, 0x0, 8) = print(args=(None, None), kw=0x0i64)
```

Listing 5.1: Examples of failed tuple reconstruction

This is because the function constructing tuples is variadic:

```
PyObject *PyTuple_Pack(Py_ssize_t n, ...)
//Return value: New reference. Part of the Stable ABI.
```

Ghidra does not exactly do a good job recognizing all the arguments to variadic functions, even when the type is set correctly.



```
                    LAB_140008a07                                   XREF[1]:     1400
140008a07 48 8b 92        MOV        RDX,qword ptr [RDX + 0xb8]=>__pyx_mstate_globa...
          b8 00 00 00
140008a0e b9 01 00        MOV        ECX,0x1
          00 00
140008a13 ff 15 47        CALL       qword ptr [->PYTHON311.DLL::PyTuple_Pack]
          eb 09 00
```

```
else {
  p_Var1 = (_object *)PyTuple_Pack(1);
  __pyx_mstate_global->__pyx_tuple__3 = p_Var1;
  if (p_Var1 != (_object *)0x0) {
```

Figure 5.2: Ghidra not recognizing all the arguments

5.3 Control flow

The restructured control flow is correct but populated with empty scopes. They are left empty because no equivalent Python definition takes place there. It can't be removed due to limitations in Ghidra API, explained in the 4.2.3. After manually removing them, however, the code resembles the original.

```
      if ((a % 2 == 0) INT_EQUAL 0x0i32):
        if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096b0, 8) , (unique, 0
            xc300, 8)):
          if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096a8, 8) , (const,
              0x0, 8)):
            goto 00104a69
          else:
            goto 00104a69
        else:
        retval = retval + 5
      else:
```

Listing 5.2: Empty scopes generated by Stackifier which contain no translated Python instructions

5.4 Examples

5.4.1 Example 1

```python
from datetime import datetime
from sys import exit

current_datetime = datetime.now()
formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted datetime:", formatted_datetime)

exit(0)
```

Listing 5.3: The sample program used throughout the thesis

```
if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009ed60, 8) , (const, 0x0, 8) ):
        import builtins as __builtins__
          &thismodule.__name__ = "__main__"
        from datetime import datetime as datetime
        from sys import exit as exit
        if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009ed90, 8) , (unique, 0xc300, 8)):
          if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009ed98, 8) , (const, 0x0, 8)):
            goto 14000917e
          else:
            goto 14000917e
        else:
        (register, 0x0, 8) = datetime.now()
        current_datetime = (register, 0x0, 8)
        if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009eda0, 8) , (unique, 0xc300, 8)):
          if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009eda8, 8) , (const, 0x0, 8)):
            goto 140009272
          else:
            goto 140009272
        else:
        (register, 0x0, 8) = current_datetime.strftime(args=(register, 0x0, 8) = __imp_PyTuple_Pack (a), kw=0x0i64)
        formatted_datetime = (register, 0x0, 8)
        if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009edb0, 8) , (unique, 0xc300, 8)):
          if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009edb8, 8) , (const, 0x0, 8)):
            goto 140009370
          else:
            goto 140009370
        else:
        (register, 0x0, 8) = print(args=("Formatted datetime:",
            formatted_datetime), kw=0x0i64)
        if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009edc0, 8) , (unique
            , 0xc300, 8)):
          if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009edc8, 8) , (
              const, 0x0, 8)):
            goto 140009465
          else:
            goto 140009465
        else:
        (register, 0x0, 8) = exit(args=(register, 0x0, 8) =
            __imp_PyTuple_Pack (a), kw=0x0i64)
        __test__ = dict()
      if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009ed60, 8) , (const, 0
          x0, 8)):
        goto 140009572
      else:
        goto 14000956d
else:
  if ((register, 0x206, 1) INT_EQUAL (ram, 0x14009ed60, 8) , (register, 0
      x8, 8)):
  else:
```

Listing 5.4: The raw decompilation output

The code is correct, except for the missing function arguments. Manually cleaned up it looks like:

```
        import builtins as __builtins__
          &thismodule.__name__ = "__main__"
        from datetime import datetime as datetime
        from sys import exit as exit

        (register, 0x0, 8) = datetime.now()
        current_datetime = (register, 0x0, 8)

        (register, 0x0, 8) = current_datetime.strftime(args=(register, 0x0
            , 8) = __imp_PyTuple_Pack (a), kw=0x0i64)
        formatted_datetime = (register, 0x0, 8)

        (register, 0x0, 8) = print(args=("Formatted datetime:",
            formatted_datetime), kw=0x0i64)
        (register, 0x0, 8) = exit(args=(register, 0x0, 8) =
            __imp_PyTuple_Pack (a), kw=0x0i64)
        __test__ = dict()
```

5.5 Example 2

```
import random

retval = 0
a = random.randint(0, 190)
if a % 2 == 0:
    retval = retval + 3
else:
    retval = retval + 5

print("Retval:", retval)
```

Listing 5.5: Another sample program, containing an if/else structure

```
if ((register, 0x206, 1) INT_EQUAL (ram, 0x109810, 8) , (const, 0x0, 8)):
   import builtins as __builtins__
    &thismodule.__name__ = "__main__"
   import random as random
   retval = 0
   if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096e0, 8) , (unique, 0xc300, 8)):
    if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096d8, 8) , (const, 0x0, 8)):
     goto 0010461d
    else:
     goto 0010461d
   else:
   (register, 0x0, 8) = random.randint(args=(0, None), kw=0x0i64)
   a = (register, 0x0, 8)
   if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096d0, 8) , (unique, 0xc300, 8)):
    if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096c8, 8) , (const, 0x0, 8)):
     goto 001046f3
    else:
     goto 001046f3
   else:
   if ((register, 0x206, 1) INT_EQUAL (ram, 0x1097c8, 8) , (register, 0x0, 8)):
   else:
    if ((register, 0x206, 1) INT_EQUAL (unique, 0xc300, 8) , (const, 0x1094a0, 8)):
     goto 0010476b
    else:
     if ((register, 0x206, 1) INT_EQUAL (unique, 0xc300, 8) , (const, 0x1092a0, 8)):
      goto 0010476b
     else:
          if ((register, 0x206, 1) INT_EQUAL (register, 0x0, 8) , (const, 0x0, 8)):
            goto 0010476b
          else:
            goto 0010476b
    if ((a % 2 == 0) INT_EQUAL 0x0i32):
      if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096b0, 8) , (unique, 0xc300, 8)):
        if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096a8, 8) , (const, 0x0, 8)):
          goto 00104a69
        else:
          goto 00104a69
      else:
      retval = retval + 5
    else:
      if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096c0, 8) , (unique, 0xc300, 8)):
        if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096b8, 8) , (const, 0x0, 8)):
          goto 001047c9
        else:
          goto 001047c9
      else:
      retval = retval + 3
    if ((register, 0x206, 1) INT_EQUAL (ram, 0x1096a0, 8) , (unique, 0xc300, 8)):
      if ((register, 0x206, 1) INT_EQUAL (ram, 0x109698, 8) , (const, 0x0, 8)):
        goto 0010487c
      else:
        goto 0010487c
    else:
    if ((register, 0x206, 1) INT_EQUAL (unique, 0x55f80, 1) , (const, 0x0, 1)):
    else:
      if ((register, 0x206, 1) INT_EQUAL (unique, 0x55f80, 1) , (const, 0x0, 1)):
      else:
        (register, 0x0, 8) = print(args=(None, None), kw=0x0i64)
        __test__ = dict()
        goto 00104c1b
  if ((register, 0x206, 1) INT_EQUAL (ram, 0x109810, 8) , (const, 0x0, 8)):
    goto 00104c1b
  else:
    goto 00104c1b
if ((register, 0x206, 1) INT_EQUAL (ram, 0x109810, 8) , (const, 0x0, 8)):
  goto 001043a4
  else:
    goto 001043a4
else:
  if ((register, 0x206, 1) INT_EQUAL (ram, 0x109810, 8) , (register, 0x38,
      8)):
    goto 001043a4
  else:
    goto 001043a4
```

Listing 5.6: The raw decompilation output

The decompiler fares pretty well in recovering it. After manually removing the dead scopes, we obtain:

```
import builtins as __builtins__
  &thismodule.__name__ = "__main__"
import random as random
retval = 0
(register, 0x0, 8) = random.randint(args=(0, None), kw=0x0i64)
a = (register, 0x0, 8)
if ((a % 2 == 0) INT_EQUAL 0x0i32):
  retval = retval + 5
else:
  retval = retval + 3
if ((register, 0x206, 1) INT_EQUAL (unique, 0x55f80, 1) , (const, 0
   x0, 1)):
else:
  if ((register, 0x206, 1) INT_EQUAL (unique, 0x55f80, 1) , (const,
     0x0, 1)):
  else:
    (register, 0x0, 8) = print(args=(None, None), kw=0x0i64)
    __test__ = dict()
```

The major problem here is, again, the recovery of the tuples. Another unpleasant aspect is the presence of empty conditional forms, which are used in the cythonized program to check for errors arising in libpython calls.

Chapter 6

Related Work

6.1 Using Ghidra microcode

River Loop security created a tool which followed use-def chains in Ghidra pCode to find vulnerabilities in function calls [1].

Ghidra-to-LLVM [1] presents an approach in reversing executables by first analyzing them in Ghidra, lifting the generated pCode to LLVM and performing further analysis on that. The authors present two approaches, one using raw pCode and one using "high" pCode extracted from a custom version of Ghidra [8].

6.2 Compilers targeting high-level languages as a backend

Both Emscripten [9] and Cherep [5] take LLVM[2] IR and produce JavaScript or WebAssembly code. They are described as compilers, not decompilers, even though they take as input a lower-level language and produce code in a higher-level language.

The Glasgow Haskell Compiler [6] converts code written in Haskell to JavaScript.

The RetDec decompiler [3] lifts binary code to LLVM IR and generates either C or a Python-like language, containing *goto* statements.

The LLVM project itself used to have a C backend. Since then, forks such as [3] have taken its place.

[1]https://github.com/toor-de-force/Ghidra-to-LLVM
[2]`https://www.llvm.org/`
[3]`https://github.com/JuliaHubOSS/llvm-cbe`

Chapter 7

Conclusion

In conclusion, cythonized code can be decompiled, but better frameworks designed with recompilation in mind, and better, more flexible intermediate languages are needed to efficiently accomplish such task.

So far Ghidra's intermediate representation was hard to work with, rigid, undocumented, and misleading. Since other people achieved better results with alternate intermediate representations and even converted Ghidra's pCode to LLVM IR, it's a legitimate question if one should continue to use this framework.

Chapter 8

Future Work

8.1 Python features

- *Custom namespaces*: So far, due to time constraints, we've explored only code defined in the global namespaces of the module. However, real Python programs have custom types, classes, and functions defined, each with their local variables and subtypes.
- *Indirect control flow*: This includes constructs such as *switch/case* statements, jump tables, exception handling via *try/catch/finally* blocks and various callback mechanisms.
- *Multiple modules*: It is possible to create and use multiple modules in the same Python project. A separation must be present in the decompiler's output.

8.2 Decompilation algorithm

The first thing on the list is to remove the dead, empty scopes. Error checks must also be removed, as they clutter the subsequent pCode with phi-nodes, hindering the data-flow analysis. For that, an alternate way to build the control flow graph, and optimize the resulting intermediate representation is needed. This can be achieved in two ways:

8.2.1 Using Ghidra's pCode

Create wrapper types over Ghidra's API, inject into or modify Ghidra's code. We need to allow pCode blocks to have their destinations changed and modifying a function's pCode at runtime then running the optimizations again.

8.3 Using alternative intermediate representations

Another idea is to lift the binary code to another intermediate representation, such as LLVM IR. It's much more flexible in interacting with the individual instructions, it uses a static-single-assignment form, which eases work and allows creating custom passes as add-ons. The downside is that it provides no framework to track strings and imported functions, read data, set types or facilitate manual reverse engineering if needed.

Bibliography

[1] Alexei Bulazel. *Working With Ghidra's P-Code To Identify Vulnerable Function Calls.* Accessed on 3rd June, 2024. URL: https://riverloopsecurity.com/blog/2019/05/pcode/.

[2] *Cython: C-Extensions for Python.* Accessed on 1st June, 2024. URL: https://cython.org/.

[3] L. Durfina, J. Kroustek, P. Matula, and P. Zemek. "A Novel Approach to Online Retargetable Machine-Code Decompilation". In: *Journal of Network and Innovative Computing* 2.1 (2014), pp. 224–232.

[4] *Embedding Cython - Cython Wiki.* Accessed on 1st June, 2024. URL: https://github.com/cython/cython/wiki/EmbeddingCython.

[5] Yuri Iozzelli. *Solving the structured control flow problem once and for all.* Accessed on 3rd June, 2024. URL: https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2.

[6] Norman Ramsey. "Beyond Relooper: recursive translation of unstructured control flow to structured control flow (functional pearl)". In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547621. URL: https://doi.org/10.1145/3547621.

[7] Lyle Ramshaw. "Eliminating go to's while preserving program structure". In: *J. ACM* 35.4 (Oct. 1988), pp. 893–920. ISSN: 0004-5411. DOI: 10.1145/48014.48021. URL: https://doi.org/10.1145/48014.48021.

[8] Toor, Tejvinder. "Decompilation of Binaries into LLVM IR for Automated Analysis". MA thesis. 2022. URL: http://hdl.handle.net/10012/17976.

[9] Alon Zakai. "Emscripten: an LLVM-to-JavaScript compiler". In: *OOPSLA Companion.* 2011. URL: https://api.semanticscholar.org/CorpusID:16356025.