



Death by a Thousand Leaks

What statically-analysing 370 Python
extensions looks like

Presented by
David Malcolm
<dmalcolm@redhat.com>

What is static analysis?

- Discovering properties of a program without running it
- Programs that analyze other programs
- Treating programs as data, rather than code
- In particular, automatically finding bugs in code

What kind of code will be
analyzed?

For this talk:

The C code of
Python extension modules

Prerequisites

- I'm going to assume basic familiarity with Python, and with either C or C++
- Hopefully you've used, debugged, or written a Python extension module in C (perhaps via SWIG or Cython)

Outline

- Intro to "cpychecker"
- How to run the tool on your own code
- How I ran the tool on lots of code
- What bugs came up frequently
- Recommendations on dealing with C and C++ from Python
- Q & A

cpychecker

**git clone **

git://git.fedorahosted.org/gcc-python-plugin.git

Docs: <http://tinyurl.com/cpychecker>

Part of my Python plugin for GCC

- 6500 lines of Python code implementing a static checker for C extension modules
- See also my PyCon US 2012 talk: Static analysis of Python extension modules using GCC
<https://us.pycon.org/2012/schedule/presentation/78/>

Reference counting



- For every object:
 - "what do I think my reference count is?" aka "ob_refcnt" (the object's view of how many pointers point to it) *versus*
 - the reality of how many pointers point to it
- As a C extension module author you must manually keep these in sync using Py_INCREF and Py_DECREF.

Reference counting



The two kinds of bugs:

- **ob_refcnt too high**
 - memory leaks (hence the title of this talk)
- **ob_refcnt too low**
 - **BOOM!!**

Checking reference counts

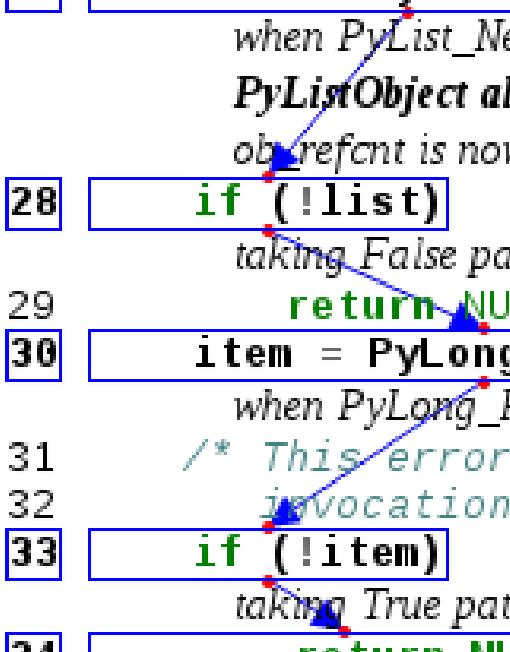
- For each path through the function and PyObject*, it determines:
 - what the reference count ought to be at the end of the function (based on how many pointers point to the object)
 - what the reference count *is*
- It will issues warnings for any that are incorrect.

File: **input.c**

Function: **test**

Error: **ob_refcnt of '*list' is 1 too high**

```
22 PyObject *
23 test(PyObject *self, PyObject *args)
24 {
25     PyObject *list;
26     PyObject *item;
27     list = PyList_New(1);
28     if (!list)
29         return NULL;
30     item = PyLong_FromLong(42);
31     /* This error handling is incorrect: it's missing an
32        invocation of Py_DECREF(list): */
33     if (!item)
34         return NULL;
35     /* This steals a reference to item; item is not leaked when we get here: */
36     PyList_SetItem(list, 0, item);
37     return list;
38 }
```



ob_refcnt of '*list' is 1 too high

was expecting final ob_refcnt to be $N + 0$ (for some unknown N)

but final ob_refcnt is $N + 1$

Limitations of the refcount checking

- purely intraprocedural
 - assumes every function returning a PyObject* returns a new reference, rather than a borrowed reference
 - (...although you can manually mark functions with non-standard behavior)
 - it knows about most of the CPython API and its rules

Limitations of the refcount checking (2)

- only tracks 0 and 1 times through any loop, to ensure that the analysis doesn't go on forever
- can be defeated by relatively simple code (turn up `--maxtrans` argument)

What it checks for (2)

It checks for the following along all of those code paths:

- Dereferencing a NULL pointer (e.g. using result of an allocator without checking the result is non-NULL)
- Passing NULL to CPython APIs that will crash on NULL

What it checks for (3)

- Usage of uninitialized local variables
- Dereferencing a pointer to freed memory
- Returning a pointer to freed memory
- Returning NULL without setting an exception

What it checks for (4)



It also does some simpler checking:

- type in calls to PyArg_ParseTuple et al
- types and NULL termination of PyMethodDef tables
- types and NULL termination of PyObject_Call{Function|Method}ObjArgs

What it doesn't check for

(patches welcome!)

- tp_traverse errors (which can mess up the garbage collector); missing it altogether, or omitting fields
- errors in GIL handling
- lock/release mismatches
- missed opportunities to release the GIL (e.g. compute-intensive functions; functions that wait on IO/syscalls)

What it can't check for



Does the code
"do the right thing"?

How to run it on your own code

```
git clone \
```

```
git://git.fedorahosted.org/gcc-python-plugin.git
```

Dependencies



(on Fedora)

```
sudo yum install \  
    gcc-plugin-devel \  
    python-devel \  
    python-six \  
    python-pygments \  
    graphviz
```

Building the checker



Building the checker:

make plugin

Checking that it's working:

make demo


```
demo.c: In function 'make_a_list_of_random_ints_badly':
demo.c:90:26: warning: Mismatching type in call to PyArg_ParseTuple with
argument 3 ("&count") had type
    "long int *" (pointing to 64 bits)
but was expecting
    "int *" (pointing to 32 bits)
for format code "i"
demo.c:102:1: warning: ob_refcnt of '*item' is 1 too high [enabled by de
demo.c:102:1: note: was expecting final ob_refcnt to be N + 1 (for some
demo.c:102:1: note: due to object being referenced by: PyListObject.ob_i
demo.c:102:1: note: but final ob_refcnt is N + 2
demo.c:97:14: note: PyLongObject allocated at:          item = PyLong_Fro
demo.c:90:26: note: when PyArg_ParseTuple() succeeds at:      if (!PyArg_
demo.c:90:8: note: taking False path at:      if (!PyArg_ParseTuple(args,
demo.c:94:10: note: reaching:      list = PyList_New(0);
demo.c:94:10: note: when PyList_New() succeeds at:      list = PyList_New
demo.c:96:5: note: when considering range: 1 <= count.0 <= 0x7fffffff at
demo.c:96:5: note: taking True path at:      for (i = 0; i < count; i++)
demo.c:97:31: note: reaching:      item = PyLong_FromLong(random());
demo.c:97:14: note: when PyLong_FromLong() succeeds at:      item = P
demo.c:97:14: note: ob_refcnt is now refs: 1 + N where N >= 0
demo.c:98:22: note: when PyList_Append() succeeds at:      PyList_App
demo.c:98:22: note: ob_refcnt is now refs: 2 + N where N >= 0
demo.c:98:22: note: '*item' is now referenced by 1 non-stack value(s): P
demo.c:96:5: note: when considering count.0 == (int)1 from demo.c:90 at:
demo.c:96:5: note: taking False path at:      for (i = 0; i < count; i++)
demo.c:101:5: note: reaching:      return list;
```

File: **input.c**

Function: **test**

Error: **ob_refcnt of '*list' is 1 too high**

```
22 PyObject *
23 test(PyObject *self, PyObject *args)
24 {
25     PyObject *list;
26     PyObject *item;
27     list = PyList_New(1);
28     if (!list)
29         return NULL;
30     item = PyLong_FromLong(42);
31     /* This error handling is incorrect: it's missing an
32        invocation of Py_DECREF(list): */
33     if (!item)
34         return NULL;
35     /* This steals a reference to item; item is not leaked when we get here: */
36     PyList_SetItem(list, 0, item);
37     return list;
38 }
```

ob_refcnt of '*list' is 1 too high

was expecting final ob_refcnt to be $N + 0$ (for some unknown N)

but final ob_refcnt is $N + 1$

Building with it

Distutils

```
CC=/path/to/built/plugin/gcc-with-cpychecker \  
python setup.py build
```

to set the environment variable

Makefiles

```
make CC=/path/to/built/plugin/gcc-with-cpychecker
```

to override the Makefile variable CC.

Let us know how you get on!

Mailing list:

- `gcc-python-plugin@lists.fedorahosted.org`
- See:
<https://fedorahosted.org/mailman/listinfo/gcc-python-plugin>

Analyze all the things!

- The goal: analyze all of the C Python extensions in a recent Linux distribution
 - Specifically: all of the Python 2 C code in Fedora 17
 - Every source rpm that builds something that links against libpython2.7
 - 370(ish) packages
- The reality:
 - Some unevenness in the data coverage, so take my numbers with a pinch of salt
 - Lots of bugfixing as I went...

Running cpychecker a lot

Scaling up to hundreds of projects:

- building via RPM
 - hides the distutils vs Makefile vs CMake etc
- "mock" builds
 - every build gets its own freshly-provisioned chroot
- Use this to reliably inject static analysis...

"mock-with-analysis"



Running checkers:

- cpychecker
- cppcheck
- clang-analyzer
- gcc warnings

<https://github.com/fedora-static-analysis/mock-with-analysis>

Scaling up (continued)

- separation of model from presentation
 - "Firehose" XML format:
 - <https://github.com/fedora-static-analysis/firehose>
- detect analyzers that fail or exceed 1 minute to run
- store the result in a database
- capture any sources mentioned in a report
- can also capture arbitrary data e.g. code metrics

```

382     free(expr); expr = NULL;
383 }
384 goto cleanup;
385
386 err:
387     error = errno;
388     py_decref(dict);
389     py_decref(tuple);
390     PyErr_SetString(PyExc_RuntimeError, strerror(error));
391 cleanup:
392     free(tmp);
393     free(rule_str);
394     free(expr);
395     errno = error;
396     return output;
397 }

```

ob_refcnt of return value is 1 too low

(emitted by cpychecker)

TODO: a detailed trace is available in the data model (not yet rendered in this report)

```

398
399 static int perform_ft_query(const apol_policy_t * policy, const options_t * opt, apol_vector_t
400 {
401     apol_filename_trans_query_t *ftq = NULL;
402     size_t i;
403     int error = 0;
404
405     if (!policy || !opt || !v) {
406         PyErr_SetString(PyExc_RuntimeError, strerror(EINVAL));
407         errno = EINVAL;

```



```
193 static PyObject *get_ipaddress(PyObject *self __unused, PyObject *args)
```

```
194 {
```

```
195     struct ifreq ifr;
```

```
196     int fd, err;
```

```
197     char *devname;
```

```
198     char ipaddr[20];
```

```
199
200     if (!PyArg_ParseTuple(args, "s", &devname))
```

Mismatching type in call to PyArg_ParseTuple with format code "s"

argument 3 ("&devname") had type "char * *" but was expecting "const char * *" for format code "s"
(emitted by cpychecker)

```
201         return NULL;
```

```
202
```

```
203     /* Setup our request structure. */
```

```
204     memset(&ifr, 0, sizeof(ifr));
```

```
205     strncpy(&ifr.ifr_name[0], devname, IFNAMSIZ);
```

```
206     ifr.ifr_name[IFNAMSIZ - 1] = 0;
```

```
207
```

```
193 static PyObject *get_ipaddress(PyObject *self __unused, PyObject *args)
```

```
194 {
```

```
195     struct ifreq ifr;
```

```
196     int fd, err;
```

```
197     const char *devname;
```

```
198     char ipaddr[20];
```

```
199
```

```
200     if (!PyArg_ParseTuple(args, "s", &devname))
```

```
201         return NULL;
```

```
202
```

```
203     /* Setup our request structure. */
```

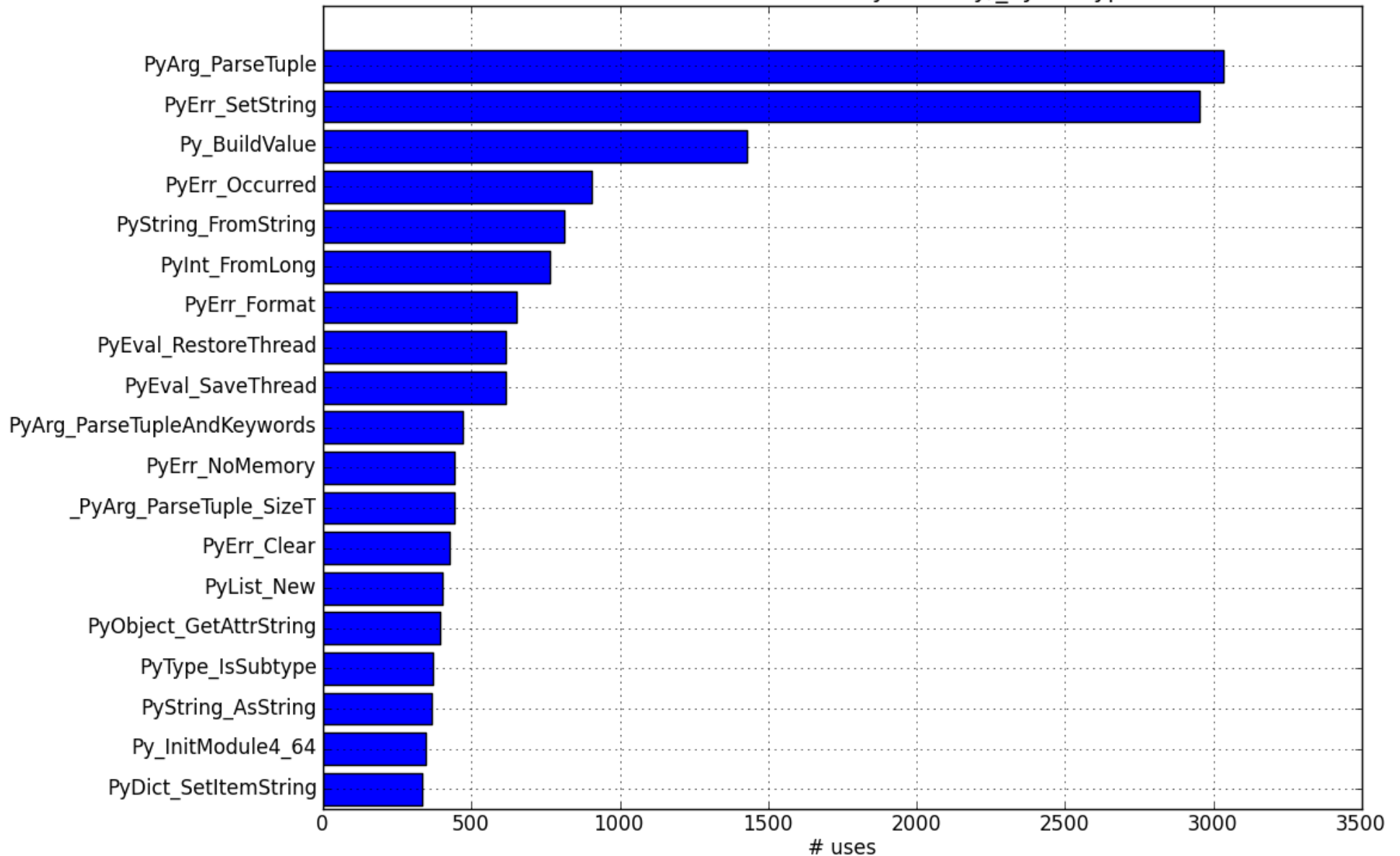
```
204     memset(&ifr, 0, sizeof(ifr));
```

```
205     strncpy(&ifr.ifr_name[0], devname, IFNAMSIZ);
```

```
206     ifr.ifr_name[IFNAMSIZ - 1] = 0;
```

```
207
```


What are the 20 most commonly used Py/_Py entrypoints?

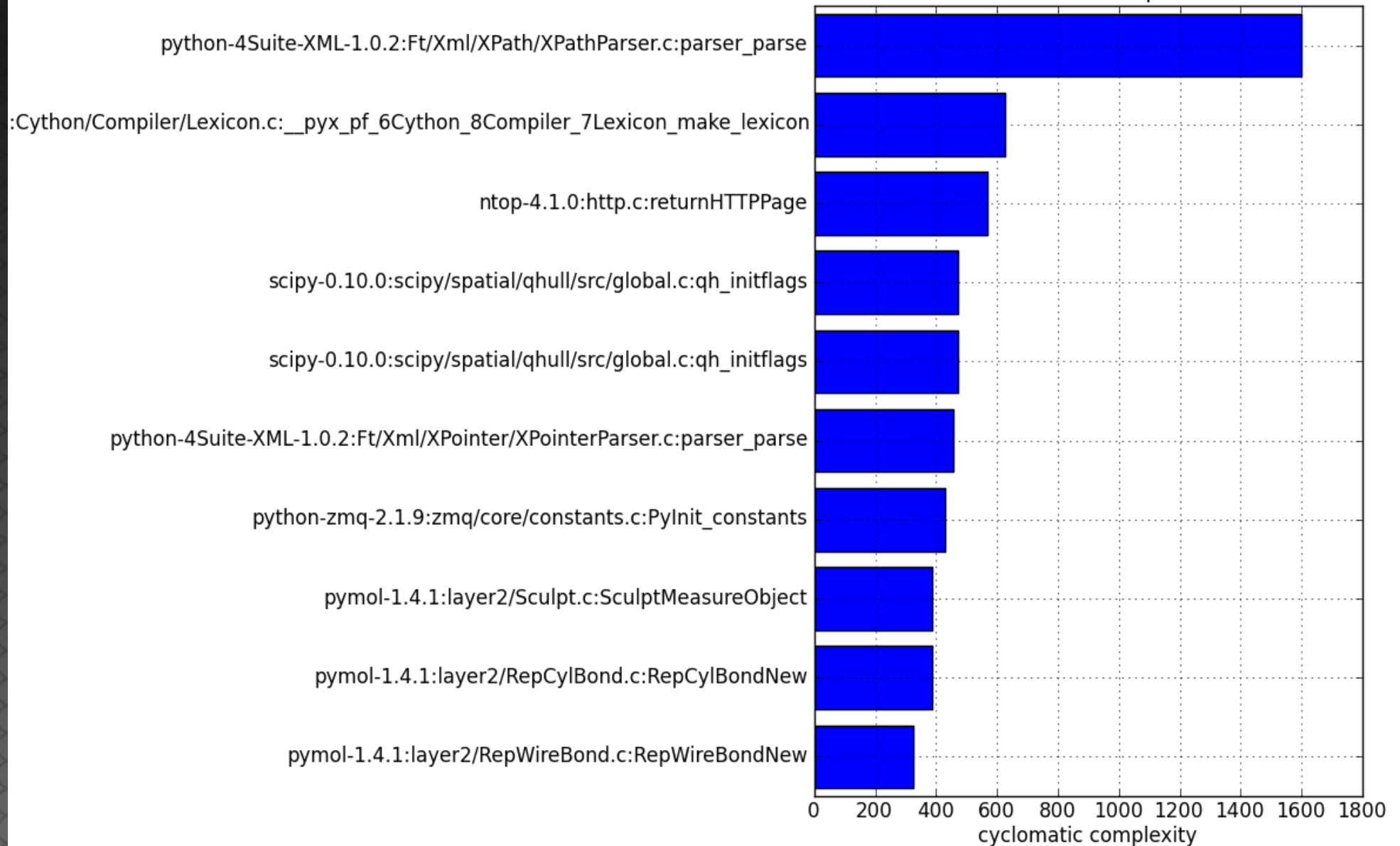


What are the least commonly used Py/_Py entrypoints?

- There are many with just 1 user, but most of these are false positives:
- about 50 actual CPython API entrypoints with just one user
- about 100 "entrypoints" due to other projects reusing the prefix

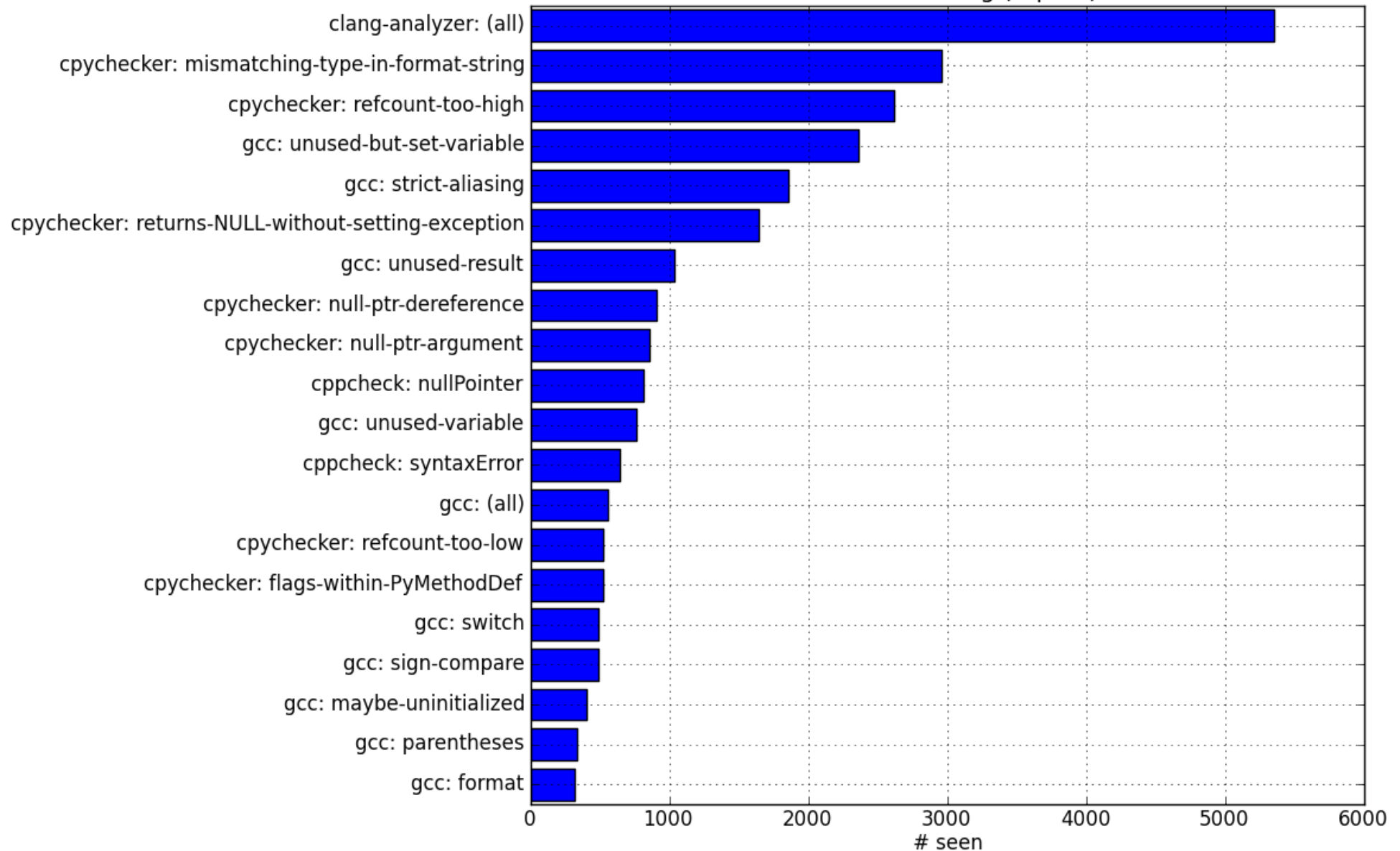
(see source code of this talk if you're interested in the data:
<https://github.com/davidmalcolm/PyCon-US-2013-Talk>)

What were the 10 most complicated functions?

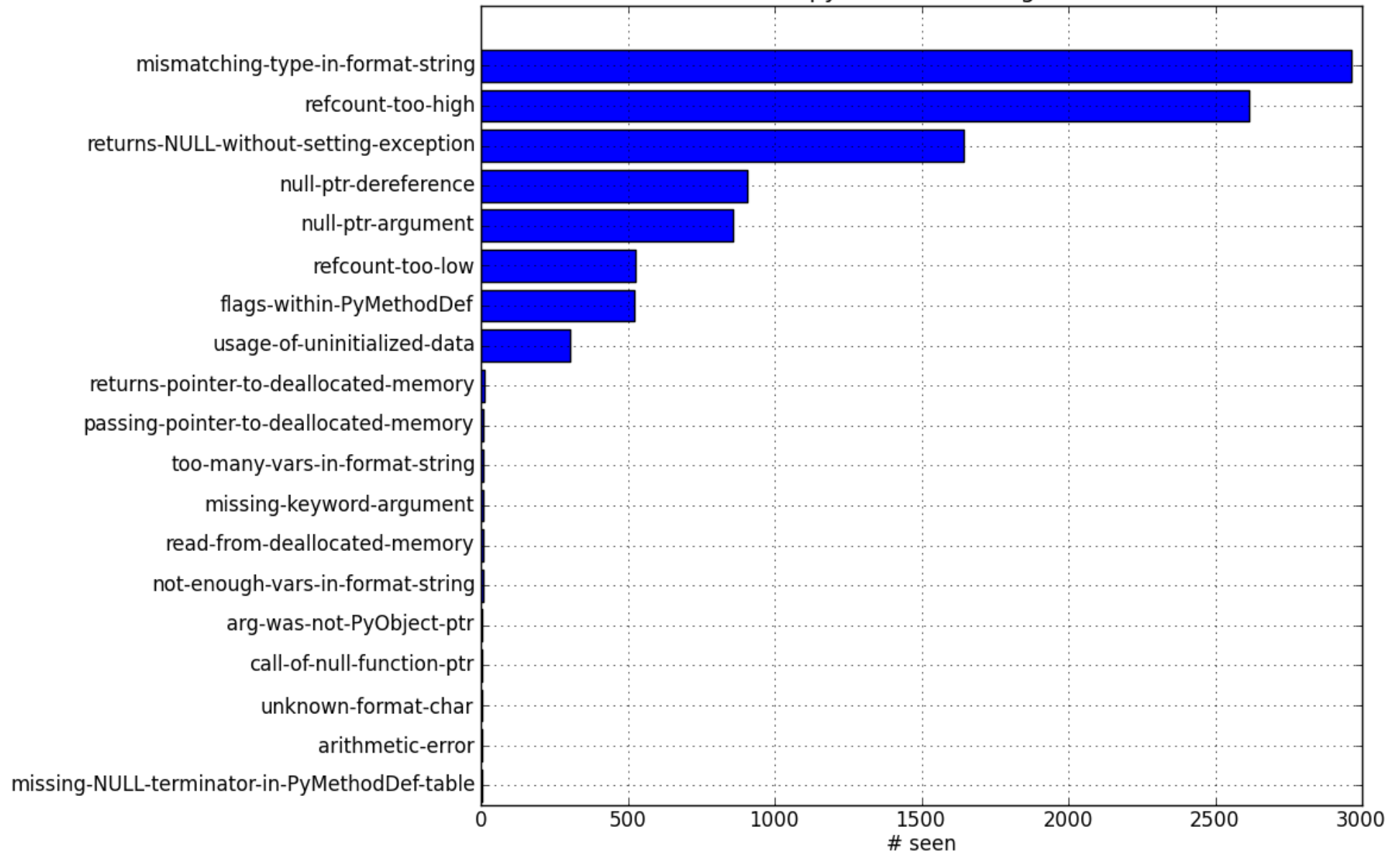


What did the analyzers
complain about?

Kinds of warning (top 20)



cpychecker warnings



Refcounting warnings



- refcount-too-high: 2614 times
- refcount-too-low: 524 times