Finding Reference-Counting Errors in Python/C Programs with Affine Analysis

Siliang Li and Gang Tan

Lehigh University, Bethlehem PA 18015, USA

Abstract. Python is a popular programming language that uses reference counting to manage heap objects. Python also has a Foreign Function Interface (FFI) that allows Python extension modules to be written in native code such as C and C++. Native code, however, is outside Python's system of memory management; therefore extension programmers are responsible for making sure these objects are reference counted correctly. This is an error prone process when code becomes complex. In this paper, we propose Pungi, a system that statically checks whether Python objects' reference counts are adjusted correctly in Python/C interface code. Pungi transforms Python/C interface code into affine programs with respect to our proposed abstractions of reference counts. Our system performs static analysis on transformed affine programs and reports possible reference counting errors. Our prototype implementation found over 150 errors in a set of Python/C programs.

Keywords: Python/C, reference counting, affine programs, static analysis.

1 Introduction

The Python programming language has become widely adopted in the software development community over the years because of many appealing features of the language itself and a robust ecosystem [1]. Similar to many other languages, Python provides a Foreign Function Interface (FFI), called the Python/C interface. The interface allows Python programs to interoperate with native modules written in C/C++. Through the interface, Python programs can reuse legacy native libraries written in C/C++ or use native code to speed up their performance-critical parts. Python provides a comprehensive set of Python/C API functions. Through these functions, native modules can create Python objects, manipulate objects, raise and handle Python exceptions, and perform other actions [2].

Another important feature of Python is its memory management. Python allocates objects on its heap. When objects are no longer in use, Python's memory manager garbage collects these objects from the heap. The standard implementation of Python uses the reference-counting algorithm. The representation of every Python object has a reference-count field. When Python code is running, the Python runtime automatically adjusts the reference counts during program execution and maintains the invariant that an object's reference count be the

R. Jones (Ed.): ECOOP 2014, LNCS 8586, pp. 80–104, 2014.

same as the number of references to the object. Specifically, the reference count of an object is incremented when there is a new reference to the object or decremented when a reference disappears. When an object's reference count becomes zero, its space is reclaimed from the heap by the garbage collector.

Native modules incorporated in a Python program, on the other hand, are outside the control of Python's garbage collector. When those native modules manipulate Python objects through the Python/C interface, reference counts are not adjusted automatically by the Python runtime and it is the native code's responsibility to adjust reference counts in a correct way (through `Py_INCREF` and `Py_DECREF`, discussed later). This is an error-prone process. Incorrect adjustments of reference counts result in classic memory errors such as memory leaks and use of dangling references.

In this paper, we describe a system called Pungi, which performs static analysis to identify reference-counting errors in native C modules of Python programs. Pungi abstracts a native module to an affine program, which models how reference counts are changed in the native module. In an affine program, the right-hand side of an assignment can only be an affine expression of the form $a_0 + \Sigma_{i=1}^{n} a_i x_i$, where a_i are constants and x_i are program variables. A previous theoretical study [3] has shown that an affine program is sufficient to model reference-count changes in the case of *shallow aliasing* (which assumes multi-level references to be non-aliases). That study, however, is mainly concerned with computational complexity and does not consider many practical issues, including function calls with parameter passing and references that escape objects' scopes. Furthermore, its proposed affine-abstraction step has not been implemented and tested for effectiveness. In fact, its affine-abstraction step is non-intuitive by requiring reversing the control flow of programs. Moreover, it does not describe how to analyze the resulting affine program to identify reference-counting errors. More detailed discussion of that work and its comparison with Pungi will be presented when we discuss the design of Pungi.

Major contributions of Pungi are described as follows:

- We propose a set of ideas that make the affine-abstraction step more complete and practical. In particular, we show how to perform affine abstraction interprocedurally and how to accommodate escaping references. We further show that the affine-abstraction step can be simplified by first performing a Static-Single Assignment (SSA) transform on the input program.
- We propose to use path-sensitive, interprocedural static analysis on the resulting affine programs to report possible reference-counting errors. We show this step is precise and efficient.
- We have built a practical reference-count analysis system that analyzes Python/C extension modules. Our system over 150 errors in 13 benchmark programs, with a modest false-positive rate of 22%.

The main limitation of Pungi is the assumption of shallow aliasing, which allows direct references to be aliases but multi-level references are assumed to reference distinct objects. For instance, if a Python program has a reference to

```
1 static PyObject* create_ntuple(PyObject *self, PyObject *args) {
2   int n, i, err;
3   PyObject *tup = NULL;
4   PyObject *item = NULL;
5   // parse args to get input number n
6   if (!PyArg_Parse(args, "(i)", &n)) return NULL;
7   tup = PyTuple_New(n);
8   if (tup == NULL) return NULL;
9   for (i=0; i<n; i++) {
10    item = PyInt_FromLong(i);
11    if (item == NULL) {Py_DECREF(tup); return NULL;}
12
13    err = PyTuple_SetItem(tup, i, item);
14    if (err) { // no need to dec-ref item
15      Py_DECREF(tup); return NULL;}
16  }
17  return tup;
18 }
```

Fig. 1. An example Python/C extension module called `ntuple` (its registration table and module initializer code are omitted)

a list object, then all objects within the list are assumed to be distinct objects. Pungi's assumption of shallow aliasing and its other assumptions may cause it to have false positives and false negatives. However, our experience shows that Pungi remains an effective tool given that it can find many reference-counting errors and its false-positive rate is moderate.

The rest of the paper is structured as follows. Sec. 2 includes the background information about the Python/C interface and reference counting. Related work is discussed in Section 3. In Sec. 4, we provide an overview of Pungi. The detailed design of Pungi is presented in Sec. 5 and 6. Pungi's implementation and a summary of its limitations are in Sec. 7. Experimental results are discussed in Sec. 8. We conclude in Sec. 9.

2 Background: The Python/C Interface and Reference Counting

The Python/C interface allows a Python program to incorporate a native library by developing a native extension module. The extension module provides a set of *native functions*. Some of the native functions are registered to be *entry native functions*, which can be imported and directly called by Python code; the rest are helper functions. An entry native function takes Python objects as input, uses Python/C API functions to create/manipulate objects, and possibly returns a Python object as the result.

Fig. 1 presents a simple C extension module called `ntuple`. It implements one function `create_ntuple`, which takes an integer `n` and constructs a tuple

(0,1,...,n-1). In more detail, references to Python objects have type "PyObject *".[1] Parameter args at line 1 is a list object, which contains the list of objects passed from Python. The call to the API function PyArg_Parse at line 6 decodes args and puts the result into integer n; format string "(i)" specifies that there should be exactly one argument, which must be an integer object. API function PyTuple_New creates a tuple with size n. The loop from line 9 to line 16 first creates an integer object using PyInt_FromLong and updates the tuple with the integer object at the appropriate index. For brevity, we have omitted the extension module's code for registering entry native functions and for initialization.

After the ntuple extension module is compiled to a dynamically linked library, it can be imported and used in Python, as shown below.

```
>>> import ntuple
>>> ntuple.create_ntuple(5)
(0, 1, 2, 3, 4)
```

2.1 Python/C Reference Counting and Its Complexities

As mentioned, native extension modules are outside the reach of Python's garbage collector. Native code must explicitly increment and decrement reference counts (we abbreviate reference counts as refcounts hereafter). Specifically,

- Py_INCREF(p) increments the refcount of the object referenced by p.
- Py_DECREF(p) decrements the refcount of the object referenced by p. When the refcount becomes zero, the object's space is reclaimed and the refcounts of all objects whose references are in object p get decremented.

Correct accounting of refcounts of objects, however, is a complex task. We next discuss the major complexities.

Control flow. Correct reference counting must be performed in all control flow paths, including those paths resulting from error conditions or interprocedural control flows. Take code in Fig. 1 as an example. At line 10, an integer object is allocated, but the allocation may fail. In the failure case, the code returns immediately, but it is also important to perform Py_DECREF on the previously allocated tup object; forgetting it would cause a memory leak. Similarly, at line 15, a Py_DECREF(tup) is necessary. Clearly, taking care of reference counts of all objects in all control-flow paths is a daunting task for programmers.

Borrowed and stolen references. It is common in native code to use the concept of *borrowed references* to save some reference-counting work. According to the Python/C manual [2], when creating a new reference to an object in a variable, "if we know that there is at least one other references to the object that lives at

[1] The Python/C interface defines type PyObject and a set of subtypes that can be used by extension code, such as PyIntObject and PyStringObject. Pungi does not distinguish these types in its analysis and treats them as synonyms. Therefore, we will just use PyObject in the rest of the paper.

least as long as our variable, there is no need to increment the reference count temporarily".

For instance, if function `foo` calls `bar` and passes `bar` a reference to an object:

```
void foo () {
  PyObject *p = PyInt_FromLong (...);
  bar (p); }
void bar (PyObject *q) { ... }
```

Within the scope of `bar`, there is one more reference (namely, `q`) to the object allocated in `foo`. However, it is safe not to increment the refcount of the object inside `bar`. The reason is that, when the control is within `bar`, we know there is at least one more reference in the caller and that reference outlives local reference `q`. Therefore, it is safe to allow more references than the refcount of the object. In this situation, the callee "borrows" the reference from the caller, meaning that the callee creates a new reference without incrementing the refcount.

Moreover, certain Python/C API functions allow callers of those functions to borrow references. For instance, `PyList_GetItem` returns a reference to an item in a list. Even though it returns a new reference to the list item, `PyList_GetItem` does not increment the refcount of the list item. This is safe when the list is not mutated before the new reference is out of scope; in this case, the reference stored in the list will outlive the new reference.[2] The Python/C reference manual lists the set of API functions with this behavior.

Dual to the situation that callers may borrow references from some API functions, certain API functions can "steal" references from the callers. For instance, in a call `PyTuple_SetItem(tuple,i,item)`, if `tuple[i]` contains an object, the object's refcount is decremented; then `tuple[i]` is set to `item`. Critically, `item`'s refcount is not incremented even though a new reference is created in the tuple. This practice is safe if we assume the `item` reference is never used after the set-item operation, which is often the case. Another behavior is that `PyTuple_SetItem(tuple,i,item)` may fail, in which case `Py_DECREF(item)` is automatically performed by the API function. This is why at line 15 in Fig. 1 there is no need to decrement the refcount on `item`.

API reference-count semantics. We have already alluded to the fact that Python/C API functions may have different effects on the refcounts of involved objects. Certain functions borrow references and certain functions steal references. Certain functions allocate objects. For instance, the calls to the API functions `PyTuple_New` and `PyInt_FromLong` in Fig. 1 allocate objects and set the refcounts of those objects to be one when allocation succeeds. And certain functions do not affect the refcounts of objects. When programmers use those API functions, they can often be confused by their effects on refcounts and make mistakes.

[2] If the list may be mutated, then the caller should increment the refcount of the retrieved object after calling `PyList_GetItem`.

All of the above factors make correct reference counting in native code extremely difficult. As a result, reference-counting errors are common in Python/C native extensions.

3 Related Work

Emmi *et al.* have used software model checking to find reference-counting errors in an OS kernel and a file system [4]. Their system's focus, assumptions, and techniques are quite different from Pungi's. The focus of their system is to find reference-counting errors in the presence of multiple threads. It assumes there is an array of reference-counted resources and assumes each resource in the array is used uniformly by a thread. Therefore, their system can use a technique called temporal case splitting to reduce the reference-counting verification of multiple resources and multiple threads to the verification of a single resource and a single thread. In the context of Python/C, however, objects passed from Python are not used uniformly by native code: an object's refcount may be adjusted differently from how other objects' refcounts are adjusted. Pungi uses an affine program to capture the effects of reference counts on objects. Another note is that the system by Emmi *et al.* assumes simple code for adjusting refcounts and has not dealt with any aliasing situation (including shallow aliasing).

Malcom has constructed a practical tool called CPyChecker [5], which is a gcc plug-in that can find a variety of errors in Python's native extension modules, including reference-counting errors. CPyChecker traverses a finite number of paths in a function and reports errors on those paths. It does not perform interprocedural analysis and ignores loops, while Pungi covers both. CPyChecker also produces wrong results when a variable is statically assigned multiple times, while Pungi uses SSA to make variables assigned only once. Experimental comparison between Pungi and CPyChecker is presented in the evaluation section.

Python/C interface code can also be generated by tools such as SWIG [6] and Cython [7]. They would reduce the number of reference-counting errors as most of the interface code is automatically generated. However, these tools do not cover all possible cases of code generation; in particular, they do not handle every feature of C/C++. As a result, a lot of interface code is still written manually in practice.

This work is an example of finding errors in Foreign Function Interface (FFI) code. Errors occur often in FFI code [8–11] because writing interface code requires resolving language differences such as memory management between two languages. Past work on improving FFIs' safety can be put into several categories. First, some systems use dynamic checking to catch errors (e.g., [12]), to enforce atomicity [13], or to isolate errors in native code so that they do not affect the host language's safety and security [14, 15]. Second, some researchers have designed new interface languages to help programmers write safer interface code (e.g.,[16]). Finally, static analysis has been used to identify specific classes of errors in FFI code, including type errors [8, 17] and exception-handling errors [11, 18]. Pungi belongs to this category and finds reference-counting errors in Python/C interface code.

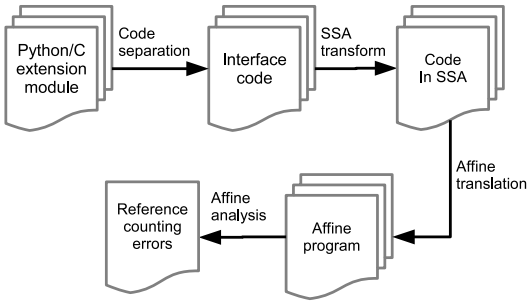

Fig. 2. An overview of Pungi

Pungi uses affine programs to abstract the reference-counting aspect of Python/C programs and performs analysis on the resulting affine programs. Affine analysis has been used in program verification in the past (e.g., [19–23]).

4 Pungi Overview

Fig. 2 shows the main steps in Pungi. It takes a Python/C extension module as input and reports reference-counting errors. Pungi analyzes only C code, but does not analyze Python code that invokes the C code.

The first step performed by Pungi is to separate *interface code* from *library code* in the extension module. As observed by a previous static-analysis system on the Java Native Interface [18], code in an FFI package can be divided into interface and library code. The library code is part of the package that belongs to a common native library. The interface code glues the host language such as Python with the native library. A native function is part of the interface code if 1) it invokes a Python/C API function, or 2) it invokes another native function that is part of the interface code. For example, the PyCrypto package has a thin layer of interface code that links Python with the underlying cryptography library. Typically, the size of interface code is much smaller than the size of library code. Therefore, Pungi performs a static analysis to separate interface code and library code so that the following steps can ignore the library code. Pungi implements a simple worklist algorithm to find functions in the interface code. If a native function does not belong to the interface code, then its execution should not have any effect on Python objects' refcounts.

After separation, *affine abstraction* converts the interface code to an affine program. The conversion is performed in two steps: Static Single Assignment (SSA) transform and affine translation. First, the SSA transform is applied on the interface code. The SSA transform makes the following affine-translation step easier to formulate; each variable is assigned only once, making it easy to track the association between variables and Python objects. In affine translation, the interface code in the SSA form is translated into an affine program. In the affine program, variables are used to track properties of Python objects, such as

their refcounts. Statements are affine operations that model how properties such as refcounts are changed in the interface code. Assertions about refcounts are also inserted into affine programs; assertion failures suggest reference-counting errors. Details of the process of affine abstraction are presented in Sec. 5.

After affine abstraction, Pungi performs an interprocedural and path-sensitive analysis that analyzes the affine program and statically checks whether assertions in the affine program hold. If an assertion might fail, a warning about a possible reference-counting error is reported. Details of affine analysis are presented in Sec. 6.

5 Affine Abstraction

For better understanding, we describe Pungi's affine abstraction in two stages. We will first present its design with the assumption that object references do not escape their scopes. We will then relax this assumption and generalize the design to allow escaping object references (*e.g.*, via return values or via a memory write to a heap data structure).

5.1 Bug Definition with Non-escaping References

One natural definition of a reference-counting error is as follows: at a program location, there is an error if the refcount of an object is not the same as the number of references to the object. However, this bug definition is too precise and an analysis based on the definition would generate too many false positives in real Python/C extension modules. This is due to the presence of borrowed and stolen references we discussed. In both cases, it is safe to make the refcount be different from the number of actual references.

Pungi's reference-counting bug definition is based on a notion of object scopes and the intuition that the expected *refcount change* of an object should be zero at the end of the object's scope (when references to the object do not escape its scope). To define an object's scope, we distinguish two kinds of objects:

- An object is a *Natively Created (NC) object* if it is created in a Python/C extension module. In Fig. 1, objects referenced by `tup` and `item` are NC objects. An NC object's scope is defined to be the immediate scope surrounding the object's creation site. For instance, the scope of the object referenced by `tup` is the function scope of `create_ntuple`.
- An object is a *Python Created (PC) object* when its reference is passed from Python to an entry native function through parameter passing. Note that we call objects whose references are passed to a native function *parameter objects*, but those parameter objects are PC objects only if that native function is an entry function. In Fig. 1, the `self` and `args` objects are PC objects. We define the scope of a PC object to be the function scope of the entry native function that receives the reference to the PC object because Pungi analyzes only native code,

```
1 void buggy_foo () {
2   PyObject * pyo = PyInt_FromLong(10);
3   if (pyo == NULL) return;
4   return;
5 }
```

Fig. 3. A contrived example of a buggy Python/C function

Definition 1. *In the case of non-escaping object references, there is a reference-counting error if, at the end of the scope of an NC or PC object, its refcount change is non-zero. If the change is greater than zero, we call it an error of reference over-counting. If the change is less than zero, we call it an error of reference under-counting.*

We next justify the bug definition. In the discussion, we use *rc* to stand for the refcount change of an object. Suppose the object is an NC object. If $rc > 0$, it results in a memory leak at the end of the scope because (1) the refcount remains positive and (2) the number of references to the object becomes zero (as object references do not escape the scope). Take the contrived code in Fig. 3 as an example. The object creation at line 2 may result in two cases. In the failure case, the object is not created and `PyInt_FromLong` returns `NULL`. In the successful case, the object is created with refcount one; in this case, the net refcount change to the object is one before returning, signaling a reference over-counting error. The correct code should have `Py_DECREF(pyo)` before line 4.

If $rc < 0$ for an NC object, then there is a use of a dangling reference because at some point of the native function execution, the refcount of the object becomes zero and the object is deallocated as a result; the next `Py_DECREF` dereferences the dangling reference.

Suppose the object is a PC object of an entry native function. We can safely assume at the beginning of the function the object's refcount is the same as the number of references to the object because the object is passed from Python, whose runtime manages refcounts automatically. If $rc > 0$ at the end of the entry native function, then after the execution of the function the object's refcount must be greater than the number of references to the object (because object references do not escape). This leads to a potential memory leak. If $rc < 0$, this leads to a dangling reference when the native function is invoked with an object whose refcount is one. Since Pungi analyzes only native code, not Python code; it has to be conservative.

One limitation of the bug definition is that it misses some dangling-reference errors that happen in the middle of native functions. For example, a native function can first decrement the refcount of a PC object and then increment the refcount. Although at the end the refcount change is zero, the object gets deallocated after the decrement if the object's original refcount is one; the following increment would use a dangling reference. This is a limitation of Pungi and we leave it to future work.

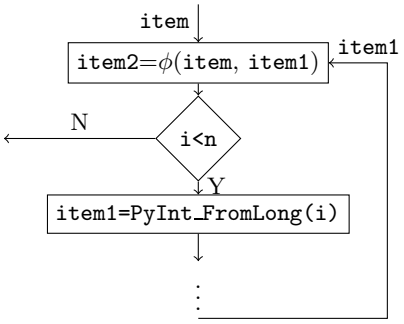

Fig. 4. Part of the control-flow graph for the code in Fig. 1 after SSA

5.2 SSA Transform

Inspired by a previous theoretical study, Pungi uses an affine program to model how refcounts are changed in the interface code of a Python/C extension module. The previous study, however, requires reversing the control-flow graph: the changes at a program location are computed based on changes that follow the location in the control-flow graph (meaning that changes for program locations later in the control-flow graph have to be computed first). The resulting affine program's control flow reverses the control flow of the original program. This process is non-intuitive and it is also unclear how to generalize it to cover function calls with parameter passing.

We observe that the fundamental reason why reversing the control-flow graph is necessary is that variables may be assigned multiple times to reference different objects. Based on this observation, Pungi simplifies the affine abstraction step by first applying the Static Single Assignment (SSA) transform to the interface code. The SSA transform inserts ϕ nodes into the program at control-flow join points and renames variables so that they are statically assigned only once. As we will show, the benefit is that Pungi does not need to reverse the control-flow graph when performing the affine-translation step; further, we can also generalize the affine translation to cover function calls with parameter passing.

Pungi's SSA transform performs transformation on only variables of type "`PyObject` *" because only Python objects are of interests to Pungi. Variables of other types are not SSA transformed. For the example in Fig. 1, variable `i` is not SSA transformed even though it is statically assigned twice. On the other hand, the `item` variable is initialized at the beginning of the code and assigned in the loop. Therefore, one ϕ node is inserted before the conditional test $i < n$ and the `item` variable is split to multiple ones. The critical parts of the control-flow graph after the SSA transform are visualized in Fig. 4

5.3 Affine Translation

The affine-translation step translates C interface code in the SSA form to an affine program that models the refcount changes of Python objects. We next

explain the intuition behind the translation, before presenting the translation algorithm.

Intuition about the affine translation. Let us assume a function takes n input references: $p_1, p_2, ..., p_n$, each of which is a reference to some Python object. Shallow aliasing allows some of these references to be aliases. For instance, p_1 and p_2 may reference the same object, in which case the refcount of the object can be changed via either p_1 or p_2.

With the assumption of shallow aliasing, Lal and Ramalingam [3] proved the following key properties:

(i) the refcount change to an object is the sum of the amount of changes made via references in $p_1, p_2, ..., p_n$ that point to the object.
(ii) the refcount change to an object via a reference is independent from the initial aliasing situation and therefore can be computed assuming an initial aliasing situation in which $p_1, p_2, ..., p_n$ are non-aliases.

We next illustrate via an example as follows:

```
p3 = p1;
Py_INCREF(p1);
Py_DECREF(p3);
Py_INCREF(p1);
Py_DECREF(p2);
```

Let us first assume `p1`, `p2`, and `p3` reference distinct objects initially. Let rc_i be the refcount change made by the program to the object that `p`i initially points to. Since it is a simple program, we can easily see that $rc_1 = 1, rc_2 = -1, rc_3 = 0$. The reason why rc_3 is zero is because `p3` is updated to be `p1` in the first statement; so there is no refcount change to the object that `p3` initially references.

Now suppose the program is actually run in an initial aliasing situation where `p1` and `p2` are aliases referencing object `a` and `p3` references a different object `b`. In this case, according to the stated properties (i) and (ii), we can compute that the refcount change to object `a` is $rc_1 + rc_2$, which is zero, and the refcount change to object `b` is rc_3, which is also zero.

The follow-up question is how to compute rc_i for an arbitrary program. The computation is modeled by an affine program, which is discussed next.

Affine program syntax. The syntax of our affine programs is presented in Fig. 5. In the syntax, we use meta-symbol x for variables and i for integer constants. An affine program consists of a set of mutually recursive functions; we assume the first function is the main function. A function declaration contains a name and a body. The body contains the declaration of a list of local variables and a block, which is a list of statements.

A statement in an affine program contains various forms of assignments, of which the right-hand sides are affine expressions. The condition c in an if-statement or a while-statement can be either a predicate, which compares a

$$
\begin{array}{rl}
(\textit{Program}) & \textit{Prog} ::= f_1; f_2; ...; f_n \\
(\textit{Function}) & f ::= \text{fname}()\{\textsf{locals } x_1, ..., x_k;\ b\} \\
(\textit{Block}) & b ::= s_1; ...; s_n \\
(\textit{Statement}) & s ::= x = i \mid x = x + i \mid x = x + y \\
 & \quad \mid \textsf{if } c \textsf{ then } \{b_1\} \textsf{ else } \{b_2\} \mid \textsf{while } c \textsf{ do } \{b\} \mid \textsf{assert } p \\
 & \quad \mid (x_1, ..., x_n) = \text{fname}() \mid \textsf{return } (x_1, ..., x_n) \\
(\textit{Condition}) & c ::= p \mid ? \\
(\textit{Predicate}) & p ::= x == i \mid x \neq i \mid x > i \mid x < i \mid x \geq i \mid x \leq i
\end{array}
$$

Fig. 5. Syntax of affine programs

variable to a constant, or a question mark. The question mark introduces nondeterminism into an affine program and is used when translating an if-statement or a while-statement with complex conditions in C code. The statement "assert p" makes an assertion about predicate p. During affine translation, the translator inserts assertions about objects' refcount changes into the affine program.

There are also function-call and function-return statements. An affine function takes zero parameters and returns a tuple. As we will discuss, a native C function with n object-reference parameters is translated to an affine function that has zero parameters and returns a tuple with n components, which are the refcount changes of the n parameter objects.

Intraprocedural affine translation. The translation from C interface code into an affine program is syntax directed, translating one function at a time. We next explain how Pungi translates a C function.

Suppose the C function takes n parameters $p_1, ..., p_n$, each of which is a reference to a Python object. We assume unique numeric labels have been given to parameter objects and object creation sites in the C function. Assume there are m labels in total, ranging from 1 to m. Among those labels, the first n labels are given to the n parameter objects and the rest to objects created in the function.

There are two important aspects about the affine translation. First, the translation maintains a *variable-object map* that maps from C variables to object labels; it tracks which object a C variable references at a program location. Second, for a Python object with label i, the affine program after translation uses a set of affine variables to track properties of the object. The most important one is the rc_i variable, which tracks the refcount change to the object. (Other affine variables will be described later.)

Fig. 6 presents the translation rules for typical C constructs. The first column of the table presents a C construct, the second column presents the updates to the variable-object map, and the last column contains the translation result.

At the function entry, the variable-object map is initialized to map from parameters to labels of parameter objects. Recall that with shallow aliasing the refcount change to an object is independent from the initial aliasing situation; this is why initially parameters are mapped to unique labels, essentially assuming they are non-aliases. In terms of translation for the function entry, refcount changes for all objects are initialized to be zero.

C construct	map updates	affine translation
function entry	forall $i \in [1..n]$ $\mathrm{map}(p_i) = i$	forall $i \in [1..m]$ $rc_i = 0$
$x = y$	$\mathrm{map}(x) = \mathrm{map}(y)$	none
`Py_INCREF`(x)		$rc_{\mathrm{map}(x)}++$
`Py_DECREF`(x)		$rc_{\mathrm{map}(x)}--$
$x =$ `PyInt_FromLong`$_l(\ldots)$	$\mathrm{map}(x) = l$	if ? then { $rc_l = 1$; $on_l = 1$ } else { $rc_l = 0$; $on_l = 0$ }
if ($x ==$ NULL) then $s1$ else $s2$		if $on_{\mathrm{map}(x)} == 0$ then { $\mathcal{T}(s1)$ } else { $\mathcal{T}(s2)$ }
`return`		forall $i \in$ OutScope($[1..m]$) assert ($rc_i == 0$) return $(rc_1, \ldots, rc_n)$
`f`$(x_1, \ldots, x_k)$		$(tmp_1, \ldots, tmp_k) =$ f(); $rc_{\mathrm{map}(x_1)}$ += tmp_1; ...; $rc_{\mathrm{map}(x_k)}$ += tmp_k;

Fig. 6. Affine translation $\mathcal{T}(-)$ for typical C constructs

Reference assignment $x = y$ results in an update to the variable-object map: afterwards, x references the same object as y. `Py_INCREF`(x) is translated to an affine statement that increments the rc variable of the object that x currently references. We use "rc_i++" as an abbreviation for $rc_i = rc_i + 1$. Similarly, `Py_DECREF`(x) is translated to a decrement on the corresponding rc variable.

The translation also translates Python/C API function calls. Such a translation required us to carefully read the Python/C reference manual about the refcount effects of API functions (and sometimes even required us to read the source code of the Python interpreter when the manual is unclear). One complication when translating API functions is the need to deal with error conditions, which are common in the Python/C interface. In the example in Fig. 3 on page 88, `PyInt_FromLong` is supposed to allocate an integer object, but the allocation may fail. The subsequent code tests whether the object is null and proceeds with two cases. Error conditions are typically signaled in the Python/C interface by returning a null reference. To deal with error conditions, Pungi introduces another affine variable for an object during translation: an object non-null variable, called the *on* variable. It is one when the object is non-null and zero when the object is null. Fig. 6 presents the translation of `PyInt_FromLong` with object label l. It is translated into a non-deterministic if-statement. In the case of an allocation success, the rc variable is set to be one and the *on* variable is also one (meaning it is non-null); in the failure case, both variables are set to be zero.

Pungi translates an if-statement in C code in a heuristic way. It recognizes a set of boolean conditions (testing for null, testing for nonnull, etc.) in the if-statement and translates those conditions accurately. Fig. 6 presents one such case when the condition is to test whether an object reference is null; the translated code tests the corresponding object's *on* variable. For complex boolean conditions, Pungi just translates them to question marks.

```
buggy_foo () {
  locals rc1, on1;
  rc1 = 0;
  if (?) {rc1 = 1; on1 = 1} else {rc1 = 0; on1 = 0};
  if (on1 == 0) { assert (rc1 == 0); return ();}
  assert (rc1 == 0); return ();
}
```

Fig. 7. Translation of the example in Fig. 3

A return statement is translated to assertions about object refcount changes followed by the returning of a tuple of refcount changes of the parameter objects. We delay the discussion why the tuple of refcount changes is returned when we discuss the interprocedural translation. An assertion is inserted for every object that is about to go outside its scope. This is according to the bug definition we discussed in Sec. 5.1. The auxiliary OutScope function returns a set of labels whose corresponding objects are about to go outside their scopes. NC (Natively Created) objects created in the function being translated belong to this set. Parameter objects are also in this set if the function is an entry native function; that is, when they are PC (Python Created) objects.

We present in Fig. 7 the translation result for the function in Fig. 3. Since the original function takes no parameters, the resulting affine function returns an empty tuple. From the affine function, we can see that the last assertion fails, which implies a reference-counting error in the original function.

We note that the SSA transform makes the presented affine-translation possible. Without the SSA, the variable-object map would possibly be updated differently in two different branches of a control-flow graph; then the translation would face the issue of how to merge two maps at a control-flow join point. After the SSA transform, an object-reference variable is statically assigned once and conflicts in variable-object maps never arise. The previous study [3] addressed the issue of variables being assigned multiple times by reversing the control flow during the affine translation. By performing the SSA transform first, Pungi simplifies the affine translation in the intraprocedural case and allows function calls that pass parameters.

Interprocedural translation. As we have seen, the affine function translated from a native function returns the refcount changes of parameter objects by assuming those parameter objects are distinct. This assumption, however, may not be true as the native function may be called with aliases and different call sites may have different aliasing situations. Fortunately, because of property (ii) in Sec. 5.3 (on page 89), it is possible to make *post-function-call refcount adjustments* according to the aliasing situation of a specific call site. The last entry in Fig. 6 describes how a function call is translated. First, the corresponding function is invoked and it returns the refcount changes of the parameter objects assuming they are distinct. After the function call, the *rc* variables of the parameter objects are adjusted according to the variable-object map of the caller.

```
void foo (PyObject *x1,
          PyObject *x2) {
  if (...) bar(x1,x1)
  else bar(x2,x2);

  return;
}

void bar (PyObject *p1,
          PyObject *p2) {
  Py_IncRef(p1); Py_DecRef(p2);
}
```

```
void foo () {
  locals rc1,on1,rc2,on2,tmp1,tmp2;
  rc1=0; rc2=0;
  if (?) {
    (tmp1,tmp2)=bar();
    rc1+=tmp1; rc1+=tmp2;
  } else {
    (tmp1,tmp2)=bar();
    rc2+=tmp1; rc2+=tmp2;
  }
  assert (rc1==0); assert (rc2==0);
  return ();
}

bar () {
  locals rc1,on1,rc2,on2;
  rc1=0; rc2=0;
  rc1++; rc2--;
  return (rc1,rc2);
}
```

Fig. 8. An example of interprocedural affine translation

Fig. 8 presents an example. On the left is some Python/C interface code, which has two functions. Function `foo` is assumed to be a native entry function. It invokes `bar` at two places. On the right of Fig. 8 is the translated affine program. Note that the post-function refcount adjustments are different for the two call sites. For the first call `f(x1,x1)`, the two refcounts are both added to `rc1`; for the second call `f(x2,x2)`, the two refcounts are both added to `rc2`.

Another note about the interprocedural translation is that, if the SSA form of a native function has ϕ nodes, then the native function is translated to multiple affine functions with one affine function created for one ϕ node. In particular, for a ϕ node, the translation finds the set of nodes in the control-flow graph that are dominated by the ϕ node and are reachable from the ϕ node without going through other ϕ nodes. This set of nodes is then translated to become the body of the affine function created for the ϕ node. Afterwards, the affine function is lifted to be a function at the global scope (that is, lambda-lifting [24]). As an example, Pungi translates the following function to exactly the same affine program on the right-hand side of Fig. 8. This is because after the SSA transform, there is a ϕ node inserted before line 4 and an additional affine function is created for that ϕ node.

```
1 void foo (PyObject *x1, PyObject *x2) {
2   PyObject *p1, *p2;
3   if (...) {p1=x1; p2=x1} else {p1=x2; p2=x2};
4   Py_IncRef(p1); Py_DecRef(p2);
5   return;
6 }
```

For the `ntuple` program in Fig. 1, since a ϕ node is inserted before the testing for loop condition (see Fig. 4), an affine function is created for the loop body; it makes a recursive call to itself because there is a control-flow edge back to the ϕ node because of the loop.

5.4 Escaping References

References to an object may escape the object's scope. In this case, the expected refcount change to the object is greater than zero. Object references may escape in several ways. A reference may escape via the return value of a function. The left-hand side of Fig. 9 presents such an example. When the integer object is successfully created, the function returns the `pyo` reference. In this case, the refcount change to the integer object is one. A reference may also escape to the heap. The code in Fig. 1 on page 82 contains such an example. At line 13, The `item` reference escapes to the heap in the tuple object when the set-item operation succeeds. In that case, the refcount change to the object created at line 10 is also one.

To deal with escaping references, we revise the bug definition as follows:

Definition 2. *There is a reference-counting error if, at the end of the scope of an NC or PC object, its refcount change is not the same as the number of times references to the object escape. If the refcount change is greater than the number of escapes, we call it an error of reference over-counting. If the change is less than the number of escapes, we call it an error of reference under-counting.*

The previous bug definition with non-escaping references is a specialization of the new definition when the number of escapes is zero. The new definition essentially uses the number of escapes to approximate the number of new references created outside the object's scope. One limitation is that an object reference may escape to the same heap location multiple times and a later escape may overwrite the references created in earlier escapes. This would result in missed errors, although this happens rarely in practice as suggested by our experience with real Python/C extension modules.

Given the new bug definition, the affine-translation step is adjusted in the following ways. First, an *escape variable*, *ev*, is introduced for each Python object and records the number of escapes. It is initialized to be zero at the beginning of a function. Second, the translator recognizes places where an object's references escape and increments the object's escape variable in the affine program by one. Third, assertions are changed to assert an objects' refcount change be the same as the number of escapes. Finally, a function not only returns the refcount changes of its parameter objects, but also returns the numbers of escapes of the parameter objects. The post-function-call adjustments adjust both the refcount changes and the numbers of escapes of the arguments.

The right-hand side of Fig. 9 presents the translated result of the code on the left. Variable `ev1` is introduced to record the number of escapes for the integer object created. This example also illustrates that the number of escapes may be different on different control-flow paths.

```
PyObject* foo () {
  PyObject *pyo=PyInt_FromLong(10);
  if (pyo==NULL) {
    return NULL;
  }
  return pyo;
}
```

```
foo () {
  locals rc1,ev1,on1;
  rc1=0; ev1=0;
  if (?) {rc1=1; on1=1}
  else {rc1=0; on1=0};
  if (on1==0) {
    assert (rc1==ev1);
    return;
  }
  if (on1==1) ev1++;
  assert (rc1==ev1);
  return (rc1,ev1);
}
```

Fig. 9. An example of escaping references

One final note is that in Pungi, with the assumption of shallow aliasing, callers of functions that return a reference are assumed to get a reference to a new object. That is, a function call that returns a reference is treated as an object-creation site.

6 Affine Analysis and Bug Reporting

The final step of Pungi is to perform analysis on the generated affine program and reports possible reference-counting errors. There are several possible analysis algorithms on affine programs, such as random interpretation [19]. Pungi adapts the ESP algorithm [25] to perform affine analysis. The major reason for choosing ESP is that it is both path-sensitive and interprocedural. The analysis has to be path sensitive to rule out impossible paths. The affine program in Fig. 9 shows a typical example. In the statement "`if (on1==0) ...`", the analysis must be able to remember the path condition `on1==0` to rule out the impossible case where `rc1==1` and `on1==1`. Without that capability, the analysis would not be able to see that the first assertion always holds. The analysis also must be interprocedural as the affine program in Fig. 8 illustrates.

ESP symbolically evaluates the program being analyzed, tracks and updates symbolic states. At every program location, it infers a set of possible symbolic states of the following form:

$$\{ \langle ps_1, es_1 \rangle, \ldots, \langle ps_n, es_n \rangle \}$$

In ESP, a symbolic state consists of a property state ps and an execution state es. The important thing about the split between property and execution states is that ESP is designed so that it is path- and context-sensitive only to the property states. Specifically, at a control-flow join point, symbolic states merge based on the property state; the execution states of all states that have the same property

```
foo () {
  locals rc1,ev1,on1;
  rc1=0; ev1=0;
  // {<[rc1=0,ev1=0], []>}
  if (?) {
    rc1=1; on1=1
    // {<[rc1=1,ev1=0], [on1=1]>}
  } else {
    rc1=0; on1=0
    // {<[rc1=0,ev1=0], [on1=0]>}
  };
  // {<[rc1=1,ev1=0], [on1=1]>, <[rc1=0,ev1=0], [on1=0]>}
  if (on1==0) {
    // {<[rc1=0,ev1=0], [on1=0]>}
    assert (rc1==ev1);
    return;
  }
  // {<[rc1=1,ev1=0], [on1=1]>}
  if (on1==1) ev1++;
  // {<[rc1=1,ev1=1], [on1=1]>}
  assert (rc1==ev1);
  return (rc1,ev1);
}
```

Fig. 10. An example of affine analysis

state are merged. By splitting property and execution states in different ways, we can control the tradeoff between efficiency and precision of the algorithm.

A particular analysis needs to decide how to split between property and execution states in ESP. We next discuss how they are defined in Pungi but leave the detailed algorithm to the ESP paper. When analyzing an affine program, Pungi's property state is the values of refcount-change variables and escape variables. The execution state is the values of all other variables.

Fig. 10 presents the analysis result at key program locations for the affine program in Fig. 9. As we can see, after the first if-statement, there are two symbolic states, representing the two branches of the if-statement. Then path sensitivity allows the analysis to eliminate impossible symbolic states after the testing of `on1==0` in the second if-statement.

We note that ESP was originally designed with a finite number of property states, while values of refcount changes and escapes can be arbitrarily large. In our implementation, we simply put a limit on those values (10 in our implementation) and used a top value when they go out of the limit.

7 Implementation and Limitations

We have built a prototype implementation of Pungi. The implementation is written in OCaml within the framework of CIL [26], which is a tool that allows

analysis and transformation of C source code. Pungi's prototype implementation cannot analyze C++ code because CIL can parse only C code. Passes are inserted into CIL to perform the separation of interface code from library code, the SSA transform, the affine translation, and the affine analysis. Our implementation of the SSA transform follows the elegant algorithm by Aycock and Horspool [27]. The total size of the implementation is around 5,000 lines of OCaml code.

Pungi also needs to identify entry native functions because assertions about parameter objects are inserted only to entry functions. Native extensions typically have a registration table to register entry functions to Python statically. Pungi searches for the table and extracts information from the table to identify entry functions. Since Python is a dynamically typed language, a native extension module can also dynamically register entry functions. Therefore, Pungi also uses some heuristics to recognize entry functions. In particular, if a function uses `PyArg_Parse` (or several other similar functions) to decode arguments, then it is treated as an entry function.

Limitations. Before we present the evaluation results of Pungi, we list its major limitations. We will discuss our plan to address some of these limitations when discussing future work. Some of these limitations have been discussed before, but we include them below for completeness.

First, Pungi assumes shallow aliasing. Whenever an object reference is retrieved from a collection object such as a list, read from a field in a struct, or returned from a function call, the reference is assumed to point to a distinct object; such a site is treated as an object-creation site.

Second, Pungi reports errors assuming Python invokes entry native functions with distinct objects. This is reflected by the fact that an assertion of the form $rc = ev$ is inserted for every parameter object of an entry native function. This assumption can be relaxed straightforwardly and please see discussion in future work.

Third, Pungi's bug definition may cause it to miss some dangling reference errors in the middle of functions, because assertions are inserted only at the end of functions.

Finally, a native extension module can call back Python functions through the Python/C API, resulting in a Ping-Pong behavior between Python and native code. An accurate analysis of such situations would require analyzing both Python and C code. On the other hand, we have not encountered such code in our experiments.

8 Evaluation

We selected 13 Python/C programs for our evaluation. These programs are common Python packages in Fedora OS and they use the Python/C interface to invoke the underlying C libraries. For instance, PyCrypto is a Python cryptography toolkit, which provides Python secure hash functions and various encryption algorithms including AES and RSA. One major reason we selected those

Table 1. Statistics about selected benchmark programs

Benchmark	Total (KLOC)	Interface code (KLOC)	Time (s)
krbV	7.0	3.7	0.78
pycrypto	16.6	7.0	1.32
pyxattr	1.0	1.0	0.09
rrdtool	31.4	0.6	0.01
dbus	93.1	7.0	0.66
gst	2.7	1.8	0.03
canto	0.3	0.2	0.001
duplicity	0.5	0.4	0.001
netifaces	1.1	1.0	0.09
pyaudio	2.9	2.7	0.03
pyOpenSSL	9.6	9.3	1.27
ldap	3.8	3.4	0.23
yum	3.0	2.4	0.20
TOTAL	173	40.5	4.7

programs for evaluation is that a previous tool, CPyChecker [5], has reported its results on those programs and we wanted to compare Pungi's results with CPyChecker's. All evaluation was run on a Ubuntu 9.10 box with 512MB memory and 2.8GHz CPU.

Table 1 lists the selected benchmarks, their sizes in terms of thousands of lines of code (KLOC), sizes of their interface code (recall that the first step Pungi performs is to separate interface from library code), and the amount of time Pungi spent on analyzing their code for reference-counting errors. The time is an average of ten runs. As we can see, Pungi is able to analyze a total of 173K lines of code in a few seconds, partly thanks to the separation between interface and library code.

The main objective in our evaluation is to know how effective our tool is in identifying the reference-counting errors as defined. This includes the number of bugs Pungi reports, the false positive rate, and the accuracy of our tool compared to CPyChecker.

Errors Found. For a benchmark program, Table 2 shows the number of warnings issued by Pungi, the numbers of true reference over- and under-counting errors, and the number of false positives. For the 13 benchmark programs, Pungi issued a total of 210 warnings, among which there are 142 true reference over-counting errors and a total of 22 true reference under-counting errors. We manually checked all true errors to the best of our ability via a two-person team. Common errors reported by both CPyChecker and Pungi have been reported to the developers by the CPyChecker author and some of those errors have been fixed in later versions of the tested benchmarks. Most of the additional true errors found by Pungi were easy to confirm manually.

Table 2. All warnings reported by Pungi, which include true reference over- and under-counting errors and false positives

Benchmark	All Warnings	Reference Over-counting	Reference Under-counting	False Positives (%)
krbV	85	74	0	11 (13%)
pycrypto	10	6	1	3 (30%)
pyxattr	4	2	0	2 (50%)
rrdtool	0	0	0	0 (0%)
dbus	3	1	0	2 (67%)
gst	30	12	13	5 (17%)
canto	6	0	4	2 (33%)
duplicity	4	2	0	2 (50%)
netifaces	8	2	1	5 (63%)
pyaudio	35	28	2	5 (14%)
pyOpenSSL	9	3	1	5 (56%)
ldap	15	11	0	4 (27%)
yum	1	1	0	0 (0%)
TOTAL	210	142	22	46 (22%)

There are 46 false positives and the overall false-positive rate is moderate, about 22%. We investigated those false positives and found most false positives are because of the following reasons:

- Object references in structs. With the assumption of shallow aliasing, Pungi treats the assignment of an object reference to a field in a struct as an escape of the reference, and treats the reading an object reference from a field of a struct as returning a reference to a new object. For example, in the following code `p` and `q` would reference two distinct objects in Pungi's analysis.

```
f->d = p;
q = f->d;
```

  As a result, Pungi loses precision when tracking refcounts in such cases. This may cause both false positives and false negatives and it contributes to the majority (22 in total) of all the false positives seen in packages such as `pycrypto` and `ldap`.
- Type casting. Pungi treats references of `PyObject` type (and its subtytpes such as `PyLongObject`, `PyIntObject`, and `PyStringObeject`) as references to Python objects. In some package code, a Python object reference is cast into another type such as an integer and then escapes to the heap. Pungi's affine translation cannot model this casting and would incorrectly issue a reference over-counting warning. 20 false positives in packages such as `gst` and `pyOpenSSL` were caused by this reason.

Table 3. Comparison of errors found between Pungi and CPyChecker

Benchmark	Pungi				CPyChecker
	Common	MA	Proc	Loop	Errors found
krbV	39	33	1	1	39
pycrypto	6	0	1	0	6
pyxattr	2	0	0	0	2
rrdtool	0	0	0	0	0
dbus	1	0	0	0	1
gst	21	2	0	2	21
canto	4	0	0	0	4
duplicity	2	0	0	0	2
netifaces	3	0	0	0	3
pyaudio	25	3	1	1	25
pyOpenSSL	1	3	0	0	1
ldap	8	3	0	0	8
yum	1	0	0	0	1
TOTAL	112	43	3	4	112

Comparison with CPyChecker. Table 3 shows the comparison of errors found between Pungi and CPyChecker. We looked into the differences and found that Pungi found all errors reported by CPyChecker. In addition, Pungi found 50 more errors than CPyChecker. The reason is because Pungi employs more precise analysis that applies the SSA and analyzes loops as well as function calls. CPyChecker's analysis is intraprocedural and ignores loops. We categorize the causes in the table. In the column Common, we put the number of errors that are reported by both Pungi and CPyChecker. Column MA (Multiple Assignments) shows the number of errors that Pungi found but missed by CPyChecker because CPyChecker's implementation cannot deal with the case when variables are statically assigned multiple times with different object references; Pungi can deal with this by the SSA transform. Column Proc shows the number of errors Pungi found but missed by CPyChecker because it cannot perform interprocedural analysis. Column Loop shows the number of errors Pungi found but missed by CPyChecker because it cannot analyze loops. The comparison shows that Pungi compares favorably to CPyChecker.

9 Conclusions and Future Work

We have described Pungi, a static-analysis tool that identifies reference-counting errors in Python/C extension modules. It translates extension code to an affine program, which is analyzed for errors of reference counting. Pungi's affine abstraction is novel in that it applies the SSA transform to simplify affine

translation and in that it can deal with the interprocedural case and escaping references. The prototype implementation found over 150 bugs in over 170K lines. We believe that Pungi makes a solid step toward statically analyzing code that uses reference counting to manage resources.

As future work, we plan to generalize Pungi to relax some of its assumptions. Pungi assumes shallow aliasing and assumes parameter objects to entry native functions are distinct objects. One possibility is to report errors for any possible aliasing situation, by adding nondeterminism into native functions. As one example, suppose an entry native function takes two parameter objects referenced by `p1` and `p2`, respectively. Suppose the function can be called either with `p1` and `p2` referencing two distinct objects or with `p1` and `p2` referencing the same object. We can insert the following code at the beginning of the native function before translation: "`if (?) {p1=p2}`", which nondeterministically initializes `p1` and `p2` for the two aliasing situations. As another example, after an object is retrieved from a list, we can nondeterministically assume the object can be a new object, or any existing object. This approach can be further improved if Python and C code are analyzed together and some alias analysis is used to eliminate impossible aliasing situations. Another possible approach to relax the shallow aliasing assumption is to keep and maintain a set of finite access paths to each Python object, as suggested by Shaham *et al.* [28].

Acknowledgement. We thank the anonymous reviewers whose comments and suggestions have helped improve the paper. This research is supported by NSF grants CCF-0915157 and CCF-1149211.

References

1. Meyerovich, L.A., Rabkin, A.S.: Empirical analysis of programming language adoption. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 1–18 (2013)
2. Python/C API reference manual (2013), `http://docs.python.org/3.3/c-api/index.html`
3. Lal, A., Ramalingam, G.: Reference count analysis with shallow aliasing. Information Processing Letters 111(2), 57–63 (2010)
4. Emmi, M., Jhala, R., Kohler, E., Majumdar, R.: Verifying reference counting implementations. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 352–367. Springer, Heidelberg (2009)
5. Malcom, D.: Cpychecker, `https://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html`
6. Beazley, D.M.: SWIG Users Manual: Version 1.1 (June 1997)
7. Cython, `http://cython.org/`
8. Furr, M., Foster, J.S.: Polymorphic type inference for the JNI. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 309–324. Springer, Heidelberg (2006)

9. Tan, G., Morrisett, G.: ILEA: Inter-language analysis across Java and C. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 39–56 (2007)
10. Kondoh, G., Onodera, T.: Finding bugs in Java Native Interface programs. In: ISSTA 2008: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 109–118. ACM, New York (2008)
11. Li, S., Tan, G.: Finding bugs in exceptional situations of JNI programs. In: 16th ACM Conference on Computer and Communications Security (CCS), pp. 442–452 (2009)
12. Lee, B., Hirzel, M., Grimm, R., Wiedermann, B., McKinley, K.S.: Jinn: Synthesizing a dynamic bug detector for foreign language interfaces. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 36–49 (2010)
13. Li, S., Liu, Y.D., Tan, G.: Native code atomicity for Java. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 2–17. Springer, Heidelberg (2012)
14. Siefers, J., Tan, G., Morrisett, G.: Robusta: Taming the native beast of the JVM. In: 17th ACM Conference on Computer and Communications Security (CCS), pp. 201–211 (2010)
15. Tan, G., Appel, A., Chakradhar, S., Raghunathan, A., Ravi, S., Wang, D.: Safe Java Native Interface. In: Proceedings of IEEE International Symposium on Secure Software Engineering, pp. 97–106 (2006)
16. Hirzel, M., Grimm, R.: Jeannie: Granting Java Native Interface developers their wishes. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 19–38 (2007)
17. Furr, M., Foster, J.: Checking type safety of foreign function calls. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 62–72 (2005)
18. Li, S., Tan, G.: JET: Exception checking in the Java Native Interface. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 345–358 (2011)
19. Gulwani, S., Necula, G.C.: Discovering affine equalities using random interpretation. In: 30th ACM Symposium on Principles of Programming Languages (POPL), pp. 74–84 (2003)
20. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: 31st ACM Symposium on Principles of Programming Languages (POPL), pp. 330–341 (2004)
21. Karr, M.: Affine relationships among variables of a program. Acta Informatica 6, 133–151 (1976)
22. Müller-Olm, M., Rüthing, O.: On the complexity of constant propagation. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 190–205. Springer, Heidelberg (2001)
23. Elder, M., Lim, J., Sharma, T., Andersen, T., Reps, T.: Abstract domains of affine relations. In: Yahav, E. (ed.) Static Analysis. LNCS, vol. 6887, pp. 198–215. Springer, Heidelberg (2011)
24. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 190–203. Springer, Heidelberg (1985)

25. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 57–68 (2002)
26. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
27. Aycock, J.: Simple generation of static single-assignment form. In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 110–124. Springer, Heidelberg (2000)
28. Shaham, R., Yahav, E., Kolodner, E.K., Sagiv, M.: Establishing local temporal heap safety properties with applications to compile-time memory management. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 483–503. Springer, Heidelberg (2003)