A Multilanguage Static Analysis of Python Programs with Native C Extensions

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

A Multilanguage Static Analysis of Python Programs with Native C Extensions*

Raphaël Monat[1], Abdelraouf Ouadjaout[1], and Antoine Miné[1,2]
`firstname.lastname@lip6.fr`

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
[2] Institut Universitaire de France, F-75005, Paris, France

Abstract. Modern programs are increasingly multilanguage, to benefit from each programming language's advantages and to reuse libraries. For example, developers may want to combine high-level Python code with low-level, performance-oriented C code. In fact, one in five of the 200 most downloaded Python libraries available on GitHub contains C code. Static analyzers tend to focus on a single language and may use stubs to model the behavior of foreign function calls. However, stubs are costly to implement and undermine the soundness of analyzers. In this work, we design a static analyzer by abstract interpretation that can handle Python programs calling C extensions. It analyses directly and fully automatically both the Python and the C source codes. It reports runtime errors that may happen in Python, in C, and at the interface. We implemented our analysis in a modular fashion: it reuses off-the-shelf C and Python analyses written in the same analyzer. This approach allows sharing between abstract domains of different languages. Our analyzer can tackle tests of real-world libraries a few thousand lines of C and Python long in a few minutes.

Keywords: Formal Methods · Static Analysis · Abstract Interpretation · Dynamic Programming Language · Multilanguage Analysis

1 Introduction

Modern programs are increasingly multilanguage. This allows developers to combine the strengths of different languages and reuse libraries written in other languages. A *host* language may call a *guest* language through an interface; this

*This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 — MOPSA.

interface is also called a *boundary*. The guest language is frequently C and is usually referred to as native code or native C. In this paper, the host language is Python, and the guest language is C. This work supports the Python API [40] as the interoperability mechanism between Python and C.[1] Using native C modules in Python is frequent as it allows writing high-level Python code, itself calling efficient C code. As a matter of fact, one in five of the 200 most downloaded Python libraries available on GitHub contains C code. Although useful, multi-language programs generate additional sources of bugs. Indeed, developers need to take into account different safety mechanisms and memory representations. Python is safe to the extent that runtime errors in pure Python programs are encapsulated into exceptions, which can be caught later on. This safety property breaks when C modules are used since a runtime error in C may irremediably terminate the program or create an inconsistent state. Python and C also have different representations. For example, Python integer objects use at least 24 bytes of memory and have unlimited precision, while C integers have fixed lengths (generally ranging from 8 to 64 bits) and can suffer from overflows.

Static analysis aims at inferring program properties (e.g., the absence of runtime errors) by analyzing programs without executing them. Static analyzers tend to focus on analyzing one language at a time. They may use stubs to model the behavior of calls to other languages. These stubs may be time-consuming to implement if written by hand. They can undermine the soundness of the analyses since the actual code is not analyzed, and the stubs may be imprecise or wrong. For example, our previous work developing a type analysis for Python [33] uses official Python type annotations (defined by PEP 484 [39]) as stubs. While this analysis tracks uncaught Python exceptions, these type annotations do not declare which exceptions may be raised, thus adding an unchecked assumption to the soundness property [33, Section 6.2].

We aim at analyzing both the native C code and the Python code (including callbacks to Python code from the native side) within the same analyzer, called Mopsa [18]. We perform a precise, flow, and context-sensitive value analysis by abstract interpretation. Our analyzer works by induction on the syntax and switches from one language to the other just as the concrete program execution does. We present a multilanguage static analysis built upon pre-existing value static analyses by abstract interpretation [9] of C [37] and Python [33, 34]. It detects runtime errors in the native C code (invalid pointer operations, invalid memory accesses, integer overflows), in the Python code (raised exceptions), and at the boundary between the languages. The underlying address allocation and numeric abstractions are shared. A few multilanguage static analyses exist, and focus mostly on analyzing Java and C ([43, 12, 13, 14, 45, 22], cf. Section 6). They compute summaries of the effects of native code on the chosen abstract property in a bottom-up fashion. Those effects are then translated into the host language, where a standard analyzer for the host language can then be used. The use of

[1]Other interoperability mechanisms such as the `ctypes` from the standard library, the `cffi` library, the `cython` project, or the `swig` project all use code using this API or generate code targetting this API. We could thus analyze this generated code.

summaries to convey the abstract meaning of functions makes it easier to rely on independent analyzers for each langage. However, the language and properties we target require precise context-sensitive value analyses that are difficult to perform bottom-up. Since Python is a dynamic programming language with a flexible semantics, it is not possible to analyze programs precisely in a context-insensitive fashion. Additionally, a precise description of the Python heap at a native call is mandatory to analyze the called C code, check for pointer errors, and infer effects. We believe the approach described in this paper is general and could be extended to other multilanguage settings, such as the analysis of Java and C through the JNI.

Contributions.

- We define a multilanguage semantics for Python programs with native C modules using the Python C API.
- We show how to lift analyses of Python and C into the multilanguage setting. The underlying address allocation and numeric abstractions are shared, paving the way for relational invariants between Python and C variables.
- We built an implementation on top of an existing static analysis platform called Mopsa that was previously used to design independent analyses for C and for Python. We added support for multilanguage C/Python programs by only adding domains modeling the boundary. We reuse the previous domains analyzing C and Python off-the-shelf. Thanks to this construction, we can detect runtime errors at the boundary, but also in the Python code and in the native C code.
- We evaluate our approach on six real-world libraries found on GitHub. We show that we can scale to libraries of 5,800 combined lines of Python and C code within five minutes.

Artifact. An artifact [35] is available alongside this article. The artifact makes Table 2 and the claim about the percentage of Python packages containing C code on Github (made in Section 1) reproducible. The example codes displayed in Figure 1 and Figure 12 are provided in the artifact, along with instructions to run our analyzer on them. The source code of the modified version of Mopsa is also included. We plan on merging our changes into the public version of Mopsa [19].

Limitations. Our concrete semantics is high-level and makes the assumption that builtin Python objects are only manipulated through the API in C (this assumption is verified by our analyzer). The garbage collection based on reference counting is not supported by our semantics. Thus, we cannot detect deallocations that are performed too soon or that are not performed at all. There is no formal soundness proof that our concrete semantics effectively models the behavior of the Python interpreter. Potential runtime errors in the API implementation modeled by our concrete semantics as builtins cannot be detected.[2] Our implementation supports relational analyses, but those do not scale to real-world examples (thus, we use intervals for the numeric abstraction by default). These limitations could be removed in future work.

Outline. We start by showing a self-contained motivating example in Section 2, giving insights on how native C modules are defined and how they work with Python. We define the concrete semantics of these multilanguage programs in Section 3, and explain the abstractions performed in Section 4. Section 5 presents our implementation and the analysis results. We discuss related work in Section 6 and conclude in Section 7.

2 An Extension Module Example

This section provides an in-depth motivating example. We show how to define a native C extension module, and how it can be used by a Python client code. We end the section by discussing which errors may happen.

When developers want to run native C code in Python, they define native C extension modules using the Python API. These modules may contain attributes, methods, and classes, just as any other Python module. However, these methods and classes are now written in C. API functions are denoted by the `Py` prefix (and written in magenta in the listings). The semantics of some of these functions are described formally in Section 3.

Counter module, viewed from Python. Our example is a C module defining a `Counter` class, alongside some client code in Python. This example is self-contained and shown in Figure 1. From a high-level point of view, the `counter` module defines a `Counter` class. Instances of `Counter` can be created (`count.py`, line 4); their internal counter can be incremented using the `incr` method, which takes an optional integer argument being the increment (lines 6-7); they also have a read-only attribute `counter` returning their current value (line 8).

Counter, viewed from C. In C, instances of `Counter` will be stored using the `CounterO` `struct`. This `struct` starts with a `PyObject ob_base` field. All Python objects are represented as `PyObjects` in C. Putting the `PyObject` as the first field in the `Counter` structure allows casting to and from Python objects.[3] The `PyObject` definition is part of the API and shown in Figure 2. Its fields are a reference counter for the garbage collector and a pointer to the class to

[2]However, half of the API supported by our implementation uses the original C implementation, which is analyzed and where runtime errors would be detected.

[3]According to the ISO C reference, “a pointer to a structure object, suitably converted, points to its initial member, and vice versa”.

count.py

```python
import counter
import random

c = counter.Counter()
p = random.randrange(128)
c.incr(2**p-1)
c.incr()
r = c.counter
```

counter.c

```c
#include <Python.h>
#include "structmember.h"

typedef struct {
  PyObject ob_base;
  int count;
} CounterO;

static PyObject*
CounterIncr(CounterO *self, PyObject *args)
{
  int i = 1;
  if(!PyArg_ParseTuple(args, "|i", &i))
    return NULL;
  self->count += i;
  Py_RETURN_NONE;
}

static int
CounterInit(CounterO *self, PyObject *args,
            PyObject *kwds)
{
  self->count = 0;
  return 0;
}

static PyMethodDef CounterMethods[] = {
 {"incr", (PyCFunction) CounterIncr,
  METH_VARARGS, ""}, {NULL}
};
```

counter.c

```c
static PyMemberDef CounterMembers[] = {
 {"counter", T_INT, offsetof(CounterO,
   count), READONLY, ""}, {NULL}
};

static PyType_Slot CounterTSlots[] = {
  {Py_tp_new, PyType_GenericNew},
  {Py_tp_init, CounterInit},
  {Py_tp_methods, CounterMethods},
  {Py_tp_members, CounterMembers}, {0, 0}
};

static PyType_Spec CounterTSpec = {
  .name = "counter.Counter",
  .basicsize = sizeof(CounterO),
  .itemsize = 0,
  .flags = Py_TPFLAGS_DEFAULT
         | Py_TPFLAGS_BASETYPE,
  .slots = CounterTSlots
};

static struct PyModuleDef countermod = {
 PyModuleDef_HEAD_INIT, .m_name = "counter",
  .m_methods = NULL, .m_size = -1
};

PyMODINIT_FUNC
PyInit_counter(void)
{
  PyObject *m = PyModule_Create(&countermod);
  if(m == NULL) return NULL;
  PyObject* CounterT =
    PyType_FromSpec(&CounterTSpec);
  if(CounterT == NULL || PyModule_AddObject(
      m, "Counter", CounterT) < 0) {
    Py_DECREF(m);
    return NULL;
  }
  return m;
}
```

Fig. 1. Example of a Python client program alongside a C counter module

```c
typedef struct PyObject {
  Py_ssize_t ob_refcnt;
  struct PyTypeObject *ob_type;
} PyObject;

typedef PyObject *(*PyCFunction)
  (PyObject *, PyObject *);
typedef int (*initproc)
  (PyObject *, PyObject *, PyObject *);

typedef struct PyMethodDef {
 const char  *ml_name; PyCFunction ml_meth;
 int  ml_flags; const char  *ml_doc;
} PyMethodDef;

typedef struct PyMemberDef {
 const char *name; int type;
 Py_ssize_t offset; int flags;
 const char *doc;
} PyMemberDef;

typedef struct PyTypeObject {
  PyObject ob_base;
  const char *tp_name;
  Py_ssize_t tp_basicsize;
  Py_ssize_t tp_itemsize;
  unsigned long tp_flags;
  struct PyMethodDef *tp_methods;
  struct PyMemberDef *tp_members;
  struct PyTypeObject *tp_base;
  PyObject *tp_dict;
  initproc tp_init;
  newfunc tp_new;
} PyTypeObject;
```

Fig. 2. Extract of Python's API header files

which it belongs. `PyTypeObject` is Python's `type` object, from which all classes derive. The second field of `CounterO` is the instance's data: an integer `count`, not directly exposed to Python.

The `Counter` class' specification is defined lines 52-59. It has three methods stored in the `Py_tp_new, Py_tp_init`, and `Py_tp_methods` fields. It also defines a special attribute member `counter` lines 41-42. The `PyTypeObject` structure is synthesized from the specification by `PyType_FromSpec` (line 72). These methods and members are lifted to become Python attributes and methods when the class is initialized (in `PyType_FromSpec`). Other fields are defined in the `PyTypeObject` structure. `tp_basicsize, tp_itemsize` define the size (in bytes) of instances. `tp_flags` is used to perform fast instance checks for builtin classes. `tp_base` points to the parent of the class. `tp_dict` is the class' dictionary used by Python to resolve attribute accesses (created during class initialization).

Module import. When executing the `import counter` statement, the C function `PyInit_counter` is called. This function starts by creating a module whose name (line 62) is `counter` with no methods attached to it (line 63). Then, the `CounterT` class is created (lines 71-72), and the class is bound to the module (lines 73-74). Python uses a reference-counting-based garbage collector, which has to be manually handled using the `Py_INCREF` and `Py_DECREF` macros in C. If no errors have been encountered, the module object is returned at the end of the function, and `counter` will now point to this module in Python.

Class initialization. The call to `PyType_FromSpec` creates the `Counter` class from its specification. The function `PyType_FromSpec` starts by creating the `PyTypeObject` and fills its fields according to the specification. This structure is then passed to `PyType_Ready` which populates the `tp_dict` field of the class. This field is the class' dictionary used by Python to resolve attribute accesses. Before this call, the attribute `counter` and the methods `__new__`, `__init__`, and `incr` do not exist on the Python side. We explain how these C functions are encapsulated into Python objects by `PyType_Ready`. The prototype of C functions for Python is `PyCFunction` (Figure 2, line 6). Some signature adaptations may be needed for specific kinds of functions. For example, initialization methods (such as `CounterInit`) return a C `int` by convention. Thus, `CounterInit` will be wrapped into a function called `wrap_init`, which behaves as a `PyCFunction`. It is then encapsulated into a builtin Python descriptor object. Upon a call to this object, this descriptor performs pre- and post-call checks (described in Section 3). Continuing our example, `wrap_init` will be stored into an instance of the builtin `wrapper_descriptor` object. These descriptors are then added to the class dictionary. Table 1 describes the three other fields.

Counter creation. When a new instance of `Counter` is created (line 4), Python starts by allocating it by calling `Counter.__new__`. This call will eventually be resolved into `PyType_GenericNew` (from `tp_new`), allocating the object and

Table 1. Python `Counter` structure summary

Attribute	Encapsulating Object	Underlying Wrapper	Underlying C definition
`__new__`	`builtin_function`	`tp_new_wrapper`	`PyType_GenericNew`
`__init__`	`wrapper_descriptor`	`wrap_init`	`CounterInit`
`incr`	`method_descriptor`	∅	`CounterIncr`
`counter`	`member_descriptor`	∅	`CounterMembers[0]`

initializing the necessary fields (such as `ob_refcnt` and `ob_type` of `PyObject`). Then, `Counter.__init__` is called and the C function `CounterInit` ends up being called. It initializes the `count` field of the `CounterO` `struct` to 0.

Counter increment. When the `incr` function is called, it is resolved through Python's attribute accesses into `CounterIncr`. `CounterIncr` uses the standard Python function prototype, corresponding to `PyCFunction` (Figure 2). Its first argument is the object instance (here, the instance stored in variable `c`), and the second argument is a tuple of the arguments passed to the function (for the call at line 6 it is a tuple of length one containing `2**p-1`, and an empty one for the second call at line 7). `PyArg_ParseTuple` is a helper C function from the Python API converting the Python arguments wrapped in the tuple into C values.[4] It uses a format string to describe the conversion. The `|` character separates mandatory arguments from the optional ones, while `i` signals a conversion from a Python integer to a C `int`. Internally, the conversion is done from a Python integer to a `long` (which may fail with an exception since Python integers are unbounded), which is then cast to an `int` if size permits (otherwise, another exception is set). In the first call to `CounterIncr`, `i` will be assigned `2**p-1` if the conversion is successful. In the second call, `i` will keep its default value, `1`. The internal value of the counter is then incremented by `i`, and then Python's `None` value is returned.

Counter access. Thanks to the complex semantics of Python, attribute accesses can actually hide calls to custom getter and setter functions through the descriptor protocol [33, Figure 6]. In our case, `PyType_Ready` takes the member declaration lines 40-43, and creates those custom getters and setters through a `member_descriptor` builtin object. The access to attribute `counter` at line 8 calls the getter of this `member_descriptor` object. This getter accesses the `count` field of the `CounterO` `struct` and converts the C integer into a Python integer. The `READONLY` flag in the declaration ensures that any call to the setter function raises an exception. These member descriptors are supported by our analysis.

[4] `Py_BuildValue` is the converse function translating C values to Python ones.

What can go wrong? Depending on the chosen value of `p` the result `r` will range from (i) the expected value ($\texttt{r} = 2^{\texttt{p}}$ when $0 \leq \texttt{p} \leq 30$), (ii) conversion errors raised as `OverflowError` exceptions (with messages depending on wheter $\texttt{p} \leq 62$), (iii) a silent integer overflow on the C side, causing a wrap-around which is an unexpected behavior for Python developers ($\texttt{r} = -2^{31}$ for $\texttt{p} = 31$). All these errors are due to different representations between the builtin values of the language. The C integer overflow does not interrupt the execution. This motivates the creation of our analysis targeting all kinds of runtime errors in both host and guest languages as well as at the boundary. Our analysis infers all reachable C and Python values, as well as raised exceptions in order to detect these runtime errors. In this example, our analyzer is able to detect all these cases. Our analyzer is also able to prove that the program is safe when `p` ranges between `0` and `30`.

Common bugs at the boundary. We refer the reader to the work of Hu and Zhang [16] for an empirical evaluation of bugs at the boundary between Python and C. The most frequent bugs happening at the boundary are:

– mismatches between a returned `NULL` and the exception being set in C (`NULL` should be returned by Python C functions if and only if an exception has been set – cf. Figure 8 in the next section),
– mismatches between the C and Python datatypes during conversion (in calls to `PyArg_ParseTuple`, `PyLong_FromLong`),
– integer overflows during conversions from arbitrary-precision Python integers to C,
– reference-counting errors (not supported by our analyzer).

3 Concrete Semantics

This section defines the semantics of the interface between Python and C. It is built upon the semantics of each language. Our goal is to delegate most of the work to the semantics of each language, each used in a black-box fashion. This delegation will also simplify our implementation as we will reuse the analyses of Python and C in a similar black-box fashion.

A key assumption of our semantics is that builtin Python objects (such as integers and dictionaries) can only be accessed from C through the API provided by the interpreter. As such, any access to the builtins through the C API can be encoded as a call back to the Python language. Thus, each language will have complementary representations of Python objects. Each language has a view of any Python object. Accesses to the other view are done by switching language. To illustrate this representation on our running example, the `int count` field from a `Counter` instance is only exposed from the C. It is possible to read the counter's value from Python. This can only be done by calling a C function performing the conversion of the C integer into a Python integer. This object is new and independent from the C integer. Hence, only C code directly dereferences the field value from the object memory. Conversely, an attribute that is dynamically added to a Python object is stored in the instance's dictionary. This dictionary is opaque from C. Accessing it through the C language using the API will ultimately be evaluated as a Python attribute access in our semantics. As illustrated in these examples, mutable parts of the objects are not *directly* available through both languages. There is thus no need to perform systematic state reductions when switching from one language to the other.

Our concrete semantics defines the operators working at the boundary. These operators allow switching from one language to another or converting the value of one language to the other. We define how Python may call C functions, and how C may perform callbacks to Python objects. We also define conversions between Python integers and C `long`s. API functions working on other builtin datatypes (such as floats, strings, lists, ...) exist and are supported by our analysis. They are similar to the integer case and are not described in this article. The definitions of these operators rely on calls to boundary functions between the two languages. These boundaries ensure that objects have the correct shape in the other language state. The boundary from C to Python also checks that only valid Python objects are passed back.

We define the state on which each semantics operates. In the following, Python-related states and expressions will be written in **green**. C-related states and expressions will be in **orange**. We reuse the states defined in the work of Monat et al. [33] for Python and Ouadjaout and Miné [37] for C. A set of heap addresses **Addr** (potentially infinite) is common to the states. Previous works [11, 33, 37] define the semantics of Python and C.

Python state (Figure 3). Python objects are split into a nominal part and a structural part. The nominal part **ObjN** can be a builtin object such as an integer (we omit other builtins for the sake of concision), a function, a class, or an instance (defined by the address of the class from which it is instantiated). The structural part **ObjS** maps attribute names to their contents' addresses. A state Σ_p consists of an environment and a heap. The environment $\mathcal{E}_p$ maps variable identifiers $\mathbf{Id}_p$ to addresses (or `LocalUndef` for local variables with an undefined value). The heap $\mathcal{H}_p$ maps addresses to objects. Given a state $\sigma_p \in \Sigma_p$, we write as $\sigma.\epsilon_p$ its environment and $\sigma.\eta_p$ its heap. Following Fromherz et al. [11], the state is additionally tagged by a flow token to handle non-local control-flow: *cur* represents the current flow on which all instructions that do not disrupt the control flow operate (e.g., assignments, but not `raise`); *exn* collects the states given by a raised exception. *exn* is indexed by the address of the exception object, so each exception will be kept separate. These continuation-tagged states are standard in abstract interpreters working by induction on the syntax to model non-local control-flow. Evaluating an expression `e` through $\mathbb{E}_p⟦ e ⟧$ in a given state yields a Python value in a new state. This value may be $\bot$ if the evaluation fails. We use letb $v, \sigma = f(e)$ in `body` as syntactic sugar for let $r, \sigma = f(e)$ in if $r \neq \bot$ then `body` else $\bot, \sigma$.

$$
\begin{aligned}
&\mathbf{ObjN} \overset{\text{def}}{=} \texttt{int}(i \in \mathbb{Z}) \cup \mathbf{Fun}(f) \\
&\cup \mathbf{Class}(c) \cup \mathbf{Inst}(a \in \mathbf{Addr}) \\
&\mathbf{ObjS} \overset{\text{def}}{=} \mathit{string} \rightharpoonup \mathbf{Addr} \\
&\mathbf{Value}_p \overset{\text{def}}{=} \mathbf{Addr} \\
&\mathcal{F}_p \overset{\text{def}}{=} \{\, \mathit{cur}, \mathit{exn}\ a \in \mathbf{Addr} \,\} \\
&\mathcal{E}_p \overset{\text{def}}{=} \mathbf{Id}_p \rightharpoonup \mathbf{Addr} \cup \texttt{LocalUndef} \\
&\mathcal{H}_p \overset{\text{def}}{=} \mathbf{Addr} \rightharpoonup \mathbf{ObjN} \times \mathbf{ObjS} \\
&\Sigma_p \overset{\text{def}}{=} \mathcal{F}_p \times \mathcal{E}_p \times \mathcal{H}_p \\
&\mathbb{E}_p[\![\, \mathit{expr} \,]\!] : \Sigma_p \to \mathbf{Value}_p{}^{\perp} \times \Sigma_p \\
&\mathbb{S}_p[\![\, \mathit{stmt} \,]\!] : \Sigma_p \to \Sigma_p
\end{aligned}
$$

Fig. 3. Concrete Python State

$$
\begin{aligned}
&\mathcal{C}\mathit{ells} \overset{\text{def}}{=} \{\, \wr b, o, t \wr \mid b \in \mathbf{Base}, t\text{: scalar} \\
&\quad \text{type}, 0 \le o \le \text{sizeof}(b) - \text{sizeof}(t) \,\} \\
&\mathit{Ptr} \overset{\text{def}}{=} (\mathbf{Base} \times \mathbb{Z}) \cup \{\, \texttt{NULL}, \texttt{invalid} \,\} \\
&\mathbf{Base} \overset{\text{def}}{=} \mathbf{Id}_c \cup \mathbf{Addr} \\
&\mathbf{Value}_c \overset{\text{def}}{=} \mathit{MNum} \cup \mathit{Ptr} \\
&\mathcal{E}_c \overset{\text{def}}{=} \mathcal{C}\mathit{ells} \rightharpoonup \mathbf{Value}_c \\
&\mathcal{H}_c \overset{\text{def}}{=} \mathbf{Addr} \rightharpoonup \mathit{ident} \times \mathbb{N} \\
&\Sigma_c \overset{\text{def}}{=} \mathcal{E}_c \times \mathcal{H}_c \\
&\mathbb{E}_c[\![\, \mathit{expr} \,]\!] : \Sigma_c \to \mathbf{Value}_c{}^{\perp} \times \Sigma_c \\
&\mathbb{S}_c[\![\, \mathit{stmt} \,]\!] : \Sigma_c \to \Sigma_c
\end{aligned}
$$

Fig. 4. Concrete C State

$$
\begin{aligned}
&\Sigma = \Sigma_p \times \Sigma_c \\
&{}^{p}\!\hookrightarrow_c : \mathbf{Value}_p \times \Sigma \to \mathbf{Value}_c \times \Sigma \\
&{}^{c}\!\hookrightarrow_p : \mathbf{Value}_c \times \Sigma \to \mathbf{Value}_p{}^{\perp} \times \Sigma \\
&\mathbb{E}_{p\times c}[\![\, \mathit{expr}_p \,]\!] : \Sigma \to \mathbf{Value}_p{}^{\perp} \times \Sigma \\
&\mathbb{E}_{p\times c}[\![\, \mathit{expr}_c \,]\!] : \Sigma \to \mathbf{Value}_c{}^{\perp} \times \Sigma
\end{aligned}
$$

Fig. 5. Combined State

C state (Figure 4). The memory is decomposed into blocks **Base** which are either variables $\mathbf{Id}_c$ or heap addresses **Addr**. Each block is decomposed into scalar elements (machine integers, floats, pointers). $\wr b, o, \tau \wr$ denotes the memory region in block b, starting at offset o and having type τ. C values $\mathbf{Value}_c$ are either machine numbers MNum, or pointers Ptr defined by their block and offset. Additionally, pointers can be `NULL` or `invalid`. The state Σ_c consists of an environment and a heap. The environment $\mathcal{E}_c$ maps scalar elements to values. The heap $\mathcal{H}_c$ maps addresses to the type of allocated resource and their size. The type of allocated resource is `Malloc` when the standard C library `malloc` is used[5]. The Python allocator (called by `PyType_GenericNew`) will create resources of type `PyAlloc`, ensuring that: (i) Python objects are well constructed by the correct allocator (ii) the C code cannot access these "opaque" objects and needs to use the API.

[5]Other resources (such as file descriptors) can also be defined [37].

Combined state (Figure 5). Two new kinds of nominal objects are added to Python: **CFun** f for Python functions defined in C, **CClass** c for Python classes defined in C (where f and c denote the name of the underlying C function or class declaration). The combined state used for the multilanguage semantics is the product of the Python and C states, written Σ. Note that each state may reference addresses originally allocated by the other language (in the running example, the Python variable `c` points to the address of the `Counter` instance, which has been allocated on the C side by `PyType_GenericNew`). In the following, we define two boundary functions converting a Python value into a C value and conversely (written ${}^{p}\hookrightarrow_{c}$ and ${}^{c}\hookrightarrow_{p}$). The multilanguage semantics $\mathbb{E}_{p\times c}[\![\cdot]\!]$ is defined over Python and C expressions. It operates over the whole state Σ and its return value matches the language of the input expression. We define the semantics of some builtins working at the boundary between Python and C, which require the whole state. For expressions not working at the boundary, the multilanguage semantics defaults to the usual Python or C semantics:

$$\mathbb{E}_{p\times c}[\![\, expr_p \,]\!](\sigma_p, \sigma_c) = \mathbb{E}_p[\![\, expr_p \,]\!](\sigma_p), \sigma_c$$
$$\mathbb{E}_{p\times c}[\![\, expr_c \,]\!](\sigma_p, \sigma_c) = \sigma_p, \mathbb{E}_c[\![\, expr_c \,]\!](\sigma_c)$$

Boundary functions. Boundary functions ensure that Python objects are correctly represented in the heap of each language. The C to Python boundary also ensures that only Python objects are passed back to Python. These functions do not convert values from one language to another. This kind of conversion is handled by builtin conversion functions such as `PyLong_AsLong`, `PyLong_FromLong` for Python integer to C long conversion. These boundary functions are lazy and shallow: (i) only objects switching languages are passed through those boundaries, (ii) an object that has already been converted does not need to be converted again (i.e., when its address is already in the other language's heap).

The boundary from Python to C is described in Figure 6. The boundary is first applied recursively to the class of the object (using the `type` operator of Python). Then, the `ob_type` field of the object is initialized to point to its class. The last step performed is to update the heap: the object has been allocated by Python, and has the size of `PyObject` (if the object is a class, it has the size of `PyTypeObject`, and we call the class initializer afterward).

The converse boundary (Figure 7) starts by checking that the value is a heap allocated Python object, allocated with resource type `PyAlloc`. It calls itself recursively on the class of the object (using the `ob_type` field in C). The Python heap is updated with the converted object.

C call from Python (Figure 8). The semantics of C function calls from Python is shown in Figure 8. The function `in_check` enforces that e_1 should be an instance of the class to which f is bound. Otherwise, a `TypeError` exception is raised. C functions callable from Python can only have two arguments (cf. the type of `PyCFunction`, Figure 2, line 6). Thus, the Python arguments are split into the first one e_1 and the other ones, bundled in a tuple. The boundary from Python to C is applied to e_1, and to the tuple containing the other arguments. Then, the C function is evaluated using the standard C semantics. Afterward, `out_check` ensures that the function returned `NULL` if and only if an exception has been set in the interpreter state. Otherwise, a `SystemError` exception is raised. Finally, the C value is passed through the boundary function.

Python call from C (Figure 9). Calls back to Python from the C code are possible using the `PyObject_CallObject` function. The first argument is the object being called. The second argument is a tuple containing all the parameters. These two arguments are first passed through the C to Python boundary. Then, we use the Python semantics to evaluate the call (the `*` operator in Python unpacks the tuple into the arguments of the variadic function). If the call is successful (i.e., the execution is normal, shown by flow token *cur*), the converse boundary function is applied. If an exception has been raised during the evalu-

```
${}^{p}\hookrightarrow_{c}(v_p, \sigma_p, \sigma_c) =$
  if $v_p \in \sigma.\eta_c$ then $(v_p, 0), \sigma_p, \sigma_c$ else
  letb $ty_p, \sigma_p = \mathbb{E}_p[\![ \mathit{type}(v_p) ]\!]\sigma_p$ in
  letb $(ty_c, 0), \sigma_p, \sigma_c = {}^{p}\hookrightarrow_{c}(ty_p, \sigma_p, \sigma_c)$ in
  let $\sigma_c = \mathbb{S}_c[\![ v_p\texttt{->ob\_type} = ty_c ]\!]\sigma_c$ in
  let $\sigma_p, \sigma_c =$
    if $\sigma_p(v_p) = \mathbf{Class}(c)$ then
      let $\sigma_c = \sigma.\epsilon_c, \sigma.\eta_c[v_p \mapsto \texttt{PyAlloc}, \mathrm{sizeof}(\texttt{PyTypeObject})]$ in
      $\mathbb{E}_{p \times c}[\![ \mathrm{PyType\_Ready}(v_p) ]\!](\sigma_p, \sigma_c)$
    else $\sigma_p, (\sigma.\epsilon_c, \sigma.\eta_c[v_p \mapsto \texttt{PyAlloc}, \mathrm{sizeof}(\texttt{PyObject})])$
   in $(v_p, 0), \sigma_p, \sigma_c$
```

Fig. 6. Python to C value boundary

```
${}^{c}\hookrightarrow_{p}(v_c, \sigma_p, \sigma_c) =$
  if $v_c \notin \mathbf{Addr} \times \{0\} \mid\mid \sigma.\eta_c(\mathrm{fst}\ v_c) \neq (\texttt{PyAlloc}, \_)$ then $\bot, \sigma_p, \sigma_c$ else
  let $v = \mathrm{fst}\ v_c$ in
  if $v \in \sigma.\eta_p$ then $v, \sigma_p, \sigma_c$ else
  letb $ty_c, \sigma_c = \mathbb{E}_c[\![ \texttt{((PyObject*)}v\texttt{)->ob\_type} ]\!]\sigma_c$ in
  letb $ty_p, \sigma_p, \sigma_c = {}^{c}\hookrightarrow_{p}(ty_c, \sigma_p, \sigma_c)$ in
  let $\sigma_p = \sigma.\epsilon_p, \sigma.\eta_p[v \mapsto \mathbf{Inst}(ty_p), \emptyset]$ in $v, \sigma_p, \sigma_c$
```

Fig. 7. C to Python value boundary

$$
\begin{array}{l}
\mathbb{E}_{p\times c}[\![\, (\textbf{CFun}\ f)(e_1, e_2, \dots, e_n) \,]\!](\sigma_p, \sigma_c) = \\
\quad \text{letb } \sigma_p = \texttt{in\_check}(\textbf{CFun}\ f, e_1, \sigma_p) \text{ in} \\
\quad \text{letb } c_1, \sigma_p, \sigma_c = {}^{p}\!\hookrightarrow_c(e_1, \sigma_p, \sigma_c) \text{ in} \\
\quad \text{letb } p_2, \sigma_p = \mathbb{E}_p[\![\, tuple(e_2, \dots, e_n) \,]\!]\sigma_p \text{ in} \\
\quad \text{letb } c_2, \sigma_p, \sigma_c = {}^{p}\!\hookrightarrow_c(p_2, \sigma_p, \sigma_c) \text{ in} \\
\quad \text{letb } c_f, \sigma_c = \mathbb{E}_c[\![\, f(c_1, c_2) \,]\!]\sigma_c \text{ in} \\
\quad \text{letb } c_f, \sigma_c = \texttt{out\_check}(c_f, \sigma_c) \text{ in} \\
\quad {}^{c}\!\hookrightarrow_p(c_f, \sigma_p, \sigma_c)
\end{array}
$$

Fig. 8. C call from Python

$$
\begin{array}{l}
\mathbb{E}_{p\times c}[\![\, \text{PyObject\_CallObject}(f, a) \,]\!](\sigma_p, \sigma_c) = \\
\quad \text{letb } f_p, \sigma_p, \sigma_c = {}^{c}\!\hookrightarrow_p(f, \sigma_p, \sigma_c) \text{ in} \\
\quad \text{letb } a_p, \sigma_p, \sigma_c = {}^{c}\!\hookrightarrow_p(a, \sigma_p, \sigma_c) \text{ in} \\
\quad \text{let } r_p, \sigma_p = \mathbb{E}_p[\![\, f_p(*a_p) \,]\!]\sigma_p \text{ in} \\
\quad \text{if } \sigma_p = (cur, \_, \_) \text{ then } {}^{p}\!\hookrightarrow_c(r_p, \sigma_p, \sigma_c) \\
\quad \text{else let } exn\ e, \epsilon_p, \eta_p = \sigma_p \text{ in} \\
\quad \text{letb } e_c, \sigma_p, \sigma_c = {}^{p}\!\hookrightarrow_c(e, (cur, \epsilon_p, \eta_p), \sigma_c) \text{ in} \\
\quad \texttt{NULL}, \sigma_p, \mathbb{S}_c[\![\, \text{PyErr\_SetNone}(e_c) \,]\!]\sigma_c
\end{array}
$$

Fig. 9. Python call from C

ation of the Python call, we revert to the *cur* flow token and pass the exception object e through the boundary. The result of the call will be `NULL`, and the exception will be set on the C side by calling `PyErr_SetNone`.

Python exceptions in C. Python exceptions may be raised from the C code using the `PyErr_SetNone` builtin. In the Python interpreter, this sets a flag in a structure describing the interpreter's state. We model this by setting a global variable `exc` with the Python object passed as an argument. Additional functions such as `PyErr_Occurred` checking if an exception has been raised and `PyErr_Clear` removing raised exceptions are modeled by accessing and setting this same global variable.

Conversion builtins of the API. We show conversion functions from C `long` to Python integers and back in Figure 10. Converting a C `long` of value v_c to a Python integer is done by calling the integer constructor in the Python semantics, and applying the boundary afterwars. To perform the other conversion, we apply the boundary function to the C value. Then, we check if the corresponding Python value v_p is an integer by looking into the Python heap. If that is the case, we check that this integer fits in a C `long` (Python integers are unbounded). Otherwise we raise an `OverflowError` and the function returns -1. A `TypeError` exception is raised and the function returns -1 if the object is not an integer.

Thanks to the definition of builtins such as `PyLong_AsLong`, other builtins calling this function can be analyzed directly using their source code from the Python interpreter's implementation. For example, when `PyArg_ParseTuple` encounters an `'i' char` in its conversion string, it executes the code shown in Figure 11.[6] As explained in our example from Section 2, it first calls `PyLong_AsLong` and converts the `long` to `int` checking for additional overflows. Our analyzer is able to analyze this piece of code directly. In our implementation, about half of

[6]`PyArg_ParseTuple` is defined as stub (just as `PyLong_AsLong` is), but the case of integers is delegated to the interpreter's implementation shown in Figure 11.

$$
\begin{array}{l}
\mathbb{E}_{p\times c}[\![\, \text{PyLong\_FromLong}(v_c) \,]\!](\sigma_p, \sigma_c) = {}^{p}\hookrightarrow_c (\mathbb{E}_p[\![\, \mathit{int}(v_c) \,]\!](\sigma_p), \sigma_c) \\
\mathbb{E}_{p\times c}[\![\, \text{PyLong\_AsLong}(v_c) \,]\!](\sigma_p, \sigma_c) = \\
\quad \text{let } v_p, \sigma_p, \sigma_c = {}^{c}\hookrightarrow_p (v_c, \sigma_p, \sigma_c) \text{ in} \\
\quad \text{if } \sigma.\eta_p = \mathit{int}(i) \text{ then} \\
\quad\quad \text{if } i \in [-2^{63}, 2^{63} - 1] \text{ then } i, \sigma_p, \sigma_c \\
\quad\quad \text{else } -1, \sigma_p, \mathbb{S}_c[\![\, \text{PyErr\_SetNone(PyExc\_OverflowError)} \,]\!]\sigma_c \\
\quad \text{else } -1, \sigma_p, \mathbb{S}_c[\![\, \text{PyErr\_SetNone(PyExc\_TypeError)} \,]\!]\sigma_c
\end{array}
$$

Fig. 10. Conversion from Python builtin integers to C long

```c
long ival = PyLong_AsLong(obj);
if(ival == -1 && PyErr_Occurred()) {
  return 0;
} else if (ival > INT_MAX) {
  PyErr_SetString(PyExc_OverflowError,
  "signed integer is greater than maximum");
  return 0;
}
```

```c
else if (ival < INT_MIN) {
  PyErr_SetString(PyExc_OverflowError,
  "signed integer is less than minimum");
  return 0;
} else {
  *result = ival;
  return 1;
}
```

Fig. 11. Python integer to C int conversion as done by `PyArg_ParseTuple`

the builtins use the original C implementation; their code is around 650 lines long.

Threats to validity. Our goal is to analyze Python programs with native C modules and detect all runtime errors that may happen. Assuming that those C modules use Python's API rather than directly modify the internal representation of builtins seems reasonable when analyzing third-party modules. This is the recommended approach for developers, as it eases maintenance of the codebase since API changes are not frequent and documented. Our analysis is able to detect if a program does not use the Python API and tries to modify a builtin Python object directly.

This concrete semantics is already high-level. A lower-level semantics where implementation details of builtins are exposed would be much more complex. It would not benefit our analysis, which aims at detecting runtime errors in the user's codebase. We have established this semantics by reading the code of the reference Python interpreter. Proving that our semantics is a sound abstraction of such lower-level semantics is left as future work.

4 Abstract Semantics

Concrete states use numeric values in different places (e.g, the C state has machine numbers, pointer offsets and resource sizes). All these values will be

```
rel_count.py
c = counter.Counter()
for i in range(randint(1, 100)):
    c.incr()
r = c.counter
assert(r == i+1)
```

Fig. 12. Example code where relationality between C and Python improves precision

centralized in a common numeric domain in the abstract state. This centralization allows expressing invariants between all those numeric variables, possibly improving the precision. We show a generic construction of the abstract multilanguage state. We assume the abstract semantics of Python and C are provided through $\mathbb{E}_p^\#[\![\cdot]\!], \mathbb{E}_c^\#[\![\cdot]\!]$. These can be instantiated in practice using previous works [37, 34]. We assume that each language's abstract state relies on an address allocation abstraction (such as the callsite abstraction or the recency abstraction [2]) and a numeric abstraction (such as intervals, octagons, ...). We write $\Sigma_u^\sharp$ the cartesian product of these two abstractions. The abstract Python (resp. C) state can then be decomposed as a product $\Sigma_p^\sharp = \tilde{\Sigma}_p^\sharp \times \Sigma_u^\sharp$ (resp. $\Sigma_c^\sharp = \tilde{\Sigma}_c^\sharp \times \Sigma_u^\sharp$). The multilanguage abstract state consists in the cartesian product of the Python and C abstract states, where the address allocation and numeric states are shared: $\Sigma_{p\times c}^\sharp = \tilde{\Sigma}_p^\sharp \times \tilde{\Sigma}_c^\sharp \times \Sigma_u^\sharp$.

Just as the concrete semantics builds upon the underlying C and Python semantics, so does our abstract semantics. The abstract semantics of the boundary operators is structurally similar to the concrete ones (each concrete operator is replaced with its abstract counterpart). We show this transformation on the abstract semantics of PyLong_FromLong (to be compared with Figure 10).

$$\mathbb{E}_{p\times c}^\#[\![\,\mathrm{PyLong\_FromLong}(v_c)\,]\!](\sigma_p^\sharp, \sigma_c^\sharp) = {}^{p}\!\hookrightarrow_c^\sharp \left(\mathbb{E}_p^\#[\![\, int(v_c)\,]\!](\sigma_p^\sharp), \sigma_c^\sharp\right)$$

Sharing the address allocation and numeric abstractions allows expressing relational invariants between the languages. In the example in Figure 12, a non-relational analysis would be able to infer that $0 \le \mathtt{i} \le 99$, but it cannot infer that the number of calls to `incr` is finite. It would thus infer that $-2^{31} \le \mathtt{r} < 2^{31}$, report an overflow error and be unable to prove the assertion at the end. Note that the value of `r` originates from the C value of the `count` field in the instance defined in `c`. With a relational analysis where C and Python variables are shared in the numeric domains, it is possible to infer that $num(@_{int}) + 1 = num(\wr @_{Counter}, 16, \mathtt{s32} \wr)$. $num(@_{int})$ is the numeric value of the integer bound to i. $num(\wr @_{Counter}, 16, \mathtt{s32} \wr)$ is the numeric value of the `Counter` instance (i.e., the value of `count` in the `Counter` `struct`, here represented as the cell [31] referenced by the `Counter` instance, at offset 16 being a 32-bit integer). Our analyzer is able to prove that the assertion holds using the octagon abstract domain [32].

Soundness. Assuming the underlying abstract semantics of Python and C are sound, the only cases left in the soundness proof are those of the operators working at the boundary. Since the abstract semantics of those operators is in point-to-point correspondence with the concrete semantics, the soundness proof is straightforward.

5 Experimental Evaluation

Implementation. We have implemented our multilanguage analysis within the open-source, publicly-available static analysis framework called Mopsa [18, 19]. A specific feature of Mopsa is to provide loosely-coupled abstract domains that the user combines through a configuration file to define the analysis. We were able to reuse off-the-shelf value analyses of C programs [37] and Python programs [34] already implemented into Mopsa. The only modification needed was to add a multilanguage domain, implementing the semantics of the operators at the boundary, just as our semantics (both concrete and abstract) do.

The configuration for the multilanguage analysis is shown in Figure 13. A configuration is a directed acyclic graph, where nodes are either domains or domain combinators. Domains can have their own local abstract state. The global abstract state is the product of these local states. Domains can also be iterators over language constructions. Each analyzed expression (or statement) flows from the top domain of the configuration to the bottom until one domain handles it. The multilanguage domain is at the top. It dispatches statements not operating at the boundary to the underlying Python or C analysis. The Python and C analyses are in a cartesian product, ensuring that when a statement goes through them, it will be handled by only one of the two sub-configurations. An example of stateful domain is "C.pointers", which keeps points-to information. "Py.exceptions" is an example of iterator domain. It handles the `raise` and `try` operators of Python. Both Python and C analyses share an underlying "universal" analysis, to which they can delegate some statements. This "universal" part provides iterators for intra- and inter-procedural statements, as well as the address allocation and numeric abstractions. The numeric abstraction displayed here only uses intervals, but it can be changed to a reduced product between a relational domain and intervals, for example.

This multilanguage domain consists in 2,500 lines of OCaml code (measured using the `cloc` tool), implementing 64 builtin functions such as the ones presented in the concrete semantics. This is small compared to the 11,700 lines of OCaml for the C analysis and 12,600 lines of OCaml for the Python analysis. These domains rely on "universal" domains representing 5,600 lines of OCaml and a common framework of 13,200 lines of OCaml. We also reuse the C implementation of 60 CPython functions as-is.

Corpus selection. In order to perform our experimental evaluation, we selected six popular Python libraries from GitHub (having in average 412 stars). These libraries are written in C and Python and do not have external dependencies. The `noise` library [10] aims at generating Perlin noise. Libraries

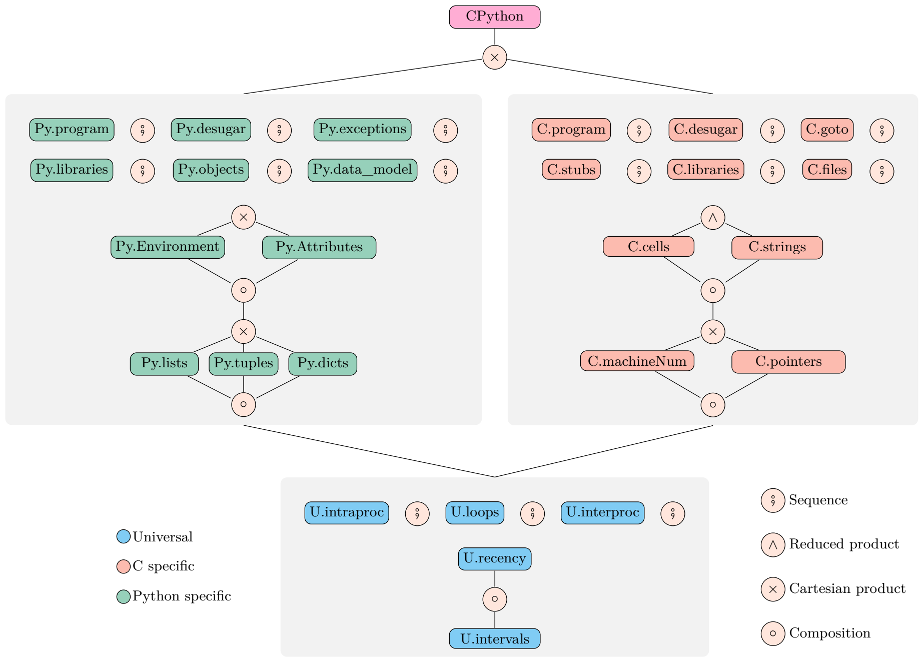

Fig. 13. Multilanguage configuration in Mopsa

`levenshtein`, `ahocorasick`, `cdistance` [15, 36, 30] implement various string-related algorithms. `llist` [17] defines linked-list objects (both single and double ones). `bitarray` [41] provides an implementation for efficient arrays of bits. Our analysis is context-sensitive in order to perform a precise value analysis. Thus, we needed some client code to analyze those libraries. We decided to analyze the tests defined by those libraries: they should cover most usecases of the library, and ensure that the transition between Python and C are frequent, which is ideal to stress-test our analysis. Some libraries (`noise`, `bitarray`, and `llist`) come with example programs with less than 50 lines of code that we analyze within 15 seconds. We have not been able to find applications with a well-defined entry point using those libraries (or they had big dependencies such as `numpy`). Our experimental evaluation thus focuses on the analysis of the libraries' tests.

Analysis results. We show the results of our analysis in Table 2. The column |C| (resp. |Py|) shows the lines of code in C (resp. Python), measured using the `cloc` tool. The Python code corresponds mainly to the tests. It may also wrap C classes in custom classes. For example, `frozenbitarray` is defined in Python, on top of the `bitarray` class. The "tests" column shows the number of tests we are able to analyze, compared to the total number of tests defined by the library. The 🕒 column shows the time taken to analyze all the tests. Columns ✓$_c$ (resp. ✓$_p$) show the selectivity of our analysis – the number of safe operations

Table 2. Analysis results of libraries using their unit tests

Library	|C|	|Py|	Tests	🕒	✓$_c$	✓$_p$	Assert.	Py$\rightsquigarrow$C
`noise`	722	675	15/15	19s	99.6% (4952)	100.0% (1738)	0/21	6.6
`ahocorasick`	3541	1336	46/92	59s	93.1% (1785)	98.0% (4937)	30/88	5.4
`levenshtein`	5441	357	17/17	1.6m	79.9% (3106)	93.2% (1719)	0/38	2.7
`cdistance`	1433	912	28/28	1.9m	95.3% (1832)	98.3% (11884)	88/207	8.6
`llist`	2829	1686	167/194	4.3m	99.0% (5311)	98.8% (30944)	235/691	51.7
`bitarray`	3244	2597	159/216	4.6m	96.3% (4496)	94.6% (21070)	97/374	14.9

compared to the total number of runtime error checks, the latter being also displayed in parentheses – performed by the analyzer for C (resp. Python). The selectivity is computed by Mopsa during the analysis. The C analysis checks for runtime errors including integer overflows, divisions by zero, invalid memory accesses and invalid pointer operations. The Python analysis checks also for runtime errors, which include the `AttributeError, TypeError, ValueError` exceptions. Runtime errors happening at the boundary are considered as Python errors since they will be raised as Python `SystemError` exceptions. The second to last column shows the number of functional properties (expressed as assertions) defined by the tests that our analyzer is able to prove correct automatically. The last column shows the number of transitions between the analyzed Python code and the C code, averaged per test.

We observe that Mopsa is able to analyze these libraries in a few minutes with high selectivity for the detection of Python and C runtime errors. Our analysis is able to detect some bugs that were previously known. For example, the `ahocorasick` module forgets to initialize some of its iterator classes, and some functions of the `bitarray` module do not set an exception when they return an erroneous flag, raising a `SystemError` exception. We have not manually checked if unknown bugs were detected by our analysis. We have instrumented Mopsa to display the number of crossings (from Python to C, or C to Python). The average number of crossings per test is shown in the last column of Table 2. The minimal number of crossings is one per test. Thus these tests seem correct to benchmark our approach since they all alternate calls to native C code and Python code.

The multilanguage analysis is limited by the current precision level of the underlying C and Python analyses but would naturally benefit immediately from any improvements in these. However, we focused on the multilanguage domains only in this study. We leave as future work the improvements required independently on the C and Python analyses for our benchmarks. We now describe a few areas where the analysis could benefit from improvements. Mopsa is unable to support some tests for now, either because they use unsupported Python libraries or because the C analysis is too imprecise to resolve some pointers. The unsupported tests of the `ahocorasick` analysis are due to imprecisions in the C analysis, which is not able to handle a complex trie data structure being stored in a dynamic array and reallocated over and over again. In `llist`, some tests use the `getrefcount` method of the `sys` which is unsupported (and related to CPython's reference-based garbage collector, which we do not support). In addition, some tests make pure-Python classes inherit from C classes: this is currently not supported in our implementation, but it is an implementation detail that will be fixed. For the `bitarray` tests, tests are unsupported because they use the unsupported `pickle` module performing object serialization, or they use the unsupported `sys.getsizeof` method, or they perform some unsupported input-output operations in Python. In addition, the C analysis is too imprecise to resolve some pointers in 18 tests.

The selectivity is lower in the C analysis of `levenshtein`, where dynamic arrays of structures are accessed in loops: the first access at `tab[i].x` may raise an alarm and continue the analysis assuming that `i` is now a valid index access. However, subsequent accesses to `tab[i].y`, `tab[i].z` will also raise alarms as the non-relational numeric domain is unable to express that `i` is a valid index access. Proving the functional properties is more challenging, and not the main goal of our analysis, which aims detecting runtime errors. For example, the assertions of the `noise` library check that the result of complex, iterative non-linear arithmetic lies in the interval $[-1, 1]$. Some assertions in the `llist` or `bitarray` library aim at checking that converting their custom container class to a list preserves the elements. Due to the smashing abstraction [3] of the Python lists, we cannot prove these assertions.

6 Related Work

Native code analysis. Some works focus on analyzing native C code in the context of language interoperability, without analyzing the host language. Tan and Croft [42] perform an empirical study of native code use in Java and provide a classification by bug patterns; a similar study has been performed by Hu and Zhang [16] for the Python/C API. Kondoh and Onodera [20] check that native calls to Java methods should handle raised exceptions using a typestate analysis. Li and Tan [23] ensure that the native control-flow is correctly interrupted when a Java exception is raised. The work of Li and Tan [24, 25] infers which Java exceptions may be raised by JNI code, allowing the exception type-safety property of Java programs to be extended to the JNI. CpyChecker [27] is a GCC plugin searching for common erroneous patterns in C code using the CPython API. Two works [26, 28] aim at detecting reference counting errors in C code using the CPython API. Brown et al. [4] define specialized analyses for specific patterns of C++ interoperability that may jeopardize type or memory safety of JavaScript. Contrary to these works, we analyze both host and guest languages.

Multilanguage semantics. The seminal work of Matthews and Findler [29] defines the first semantics of multilanguage systems, using the notion of boundaries to model conversion between languages. Buro and Mastroeni [7] generalize this construction using an algebraic framework. We use a similar notion of boundary in our concrete semantics.

Multilanguage analyses. Buro et al. [6] define a theory based on abstract interpretation to combine analyses of different languages, and show how to lift the soundness property to the multilanguage setting. They provide an example of multilanguage setting where they combine a toy imperative language with a bit-level arithmetic language. The notion of boundary functions used in their work performs a full translation from the state of one language to the other. Our semantics works on the product of the states, although it can be seen as an abstraction of the semantics of C and Python, where the boundary performs a full state conversion (but the boundary from Python to C would be a concretization). From an implementation standpoint, our approach avoids costly state conversions at the boundary and allows sharing some abstract domains.

Chipounov et al. [8] perform symbolic execution of binaries, thus avoiding language considerations. Their approach is extended by the work of Bucur et al. [5], which supports any interpreted language by performing symbolic execution over the interpreter. Our approach is more costly to implement since we do not automatically lift the interpreter's code to obtain our analyzer. Thanks to its higher-level, we think our approach should be more precise and efficient.

The next works perform multilanguage analyses by translating specific effects of what native functions do (they usually generate a summary using a bottom-up analysis) to the host language. This allows removing the native code and use existing analyses of the host language. Tan and Morrisett [43] compile C code into an extended JVML syntax form, allowing the use of the bug-finding tool Jlint afterwards. Furr and Foster [12, 13, 14] perform inference of OCaml and Java types in C FFI code, which they crosscheck with the types used in the client OCaml/Java code. They assume there are no out-of-bounds accesses and no type casting in the C code. An inter-language, bottom-up taint analysis for Java and native binary code in the setting of Android applications is proposed by Wei et al. [45]. Lee et al. [22] aim at detecting wrong forein function calls and mishandling of Java exceptions in Java/JNI code. They extract summaries of the Java callbacks and field accesses from the JNI code using Infer, transform these summaries into Java code, and call the FlowDroid analyzer on the whole. Contrary to these works, our analyzer supports both languages, and it switches between languages just as the real code does. The properties we target require precise context-sensitive value analyses that are diffcult to perform bottom-up.

Library analyses. Previous work aim at analyzing libraries with no access to their client code [1, 38] using a "most-general client". The work of Kristensen and Møller [21] refines the notion of most-general client in the setting of dynamic programming languages. However, it focuses on libraries where functions are typed. Python libraries are not explicitly typed. Extending their work to our untyped, multilanguage setting is left as future work.

7 Conclusion

This article presents a multilanguage analysis able to infer runtime errors in Python code using native C extensions. Our analyzer is able to reuse value analyses of Python and C off-the-shelf. It shares the address allocation and numeric abstractions between the Python and C abstract domains. We are able to analyze within a few minutes real-world Python libraries written in C and having up to 5,800 lines of code. To the best of our knowledge, we have implemented the first static analyzer able to soundly detect runtime errors in multilanguage programs.

Future work includes extending our implementation to analyze the standard Python library and large Python applications. This will require handling more dependencies, having a relational analysis that scales, and addressing the precision limitations of the underlying C and Python analyses. We plan to instrument our implementation to verify (or infer) type annotations of the standard library provided in the typeshed [44] project. It would also be interesting to target multilanguage-specific safety properties (such as correct garbage collection counts). Another future work is to try our approach in other multilanguage settings (such as Java/C).

Bibliography

[1] Allen, N., Krishnan, P., Scholz, B.: Combining type-analysis with points-to analysis for analyzing Java library source-code. In: SOAP@PLDI, ACM (2015)

[2] Balakrishnan, G., Reps, T.W.: Recency-abstraction for heap-allocated storage. In: SAS, Springer (2006)

[3] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The Essence of Computation, Lecture Notes in Computer Science, vol. 2566, pp. 85–108, Springer (2002)

[4] Brown, F., Narayan, S., Wahby, R.S., Engler, D.R., Jhala, R., Stefan, D.: Finding and preventing bugs in JavaScript bindings. In: SP, IEEE Computer Society (2017)

[5] Bucur, S., Kinder, J., Candea, G.: Prototyping symbolic execution engines for interpreted languages. In: ASPLOS, pp. 239–254, ACM (2014)

[6] Buro, S., Crole, R.L., Mastroeni, I.: On multi-language abstraction - towards a static analysis of multi-language programs. In: SAS, Springer (2020)

[7] Buro, S., Mastroeni, I.: On the multi-language construction. In: ESOP, Springer (2019)

[8] Chipounov, V., Kuznetsov, V., Candea, G.: S2E: a platform for in-vivo multi-path analysis of software systems. In: ASPLOS, pp. 265–278, ACM (2011)

[9] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM (1977)

[10] Duncan, C.: Native-code and shader implementations of perlin noise for Python. `https://github.com/caseman/noise` (2021), accessed: 2021-04

[11] Fromherz, A., Ouadjaout, A., Miné, A.: Static value analysis of Python programs by abstract interpretation. In: NFM, Springer (2018)

[12] Furr, M., Foster, J.S.: Checking type safety of foreign function calls. In: PLDI, ACM (2005)

[13] Furr, M., Foster, J.S.: Polymorphic type inference for the JNI. In: ESOP, Springer (2006)

[14] Furr, M., Foster, J.S.: Checking type safety of foreign function calls. ACM Trans. Program. Lang. Syst. (4) (2008)

[15] Haapala, A., Määttä, E., CD, J., Ohtamaa, M., Necas, D.: Levenshtein Python C extension module. `https://github.com/ztane/python-Levenshtein/` (2021), accessed: 2021-04

[16] Hu, M., Zhang, Y.: The Python/C API: evolution, usage statistics, and bug patterns. In: SANER, IEEE (2020)

[17] Jakubek, A., Gałczyński, R.: Linked lists for CPython. `https://github.com/ajakubek/python-llist` (2021), accessed: 2021-04

[18] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, Springer (2019)

[19] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: MOPSA: Modular Open Platform for Static Analysis. `https://gitlab.com/mopsa/mopsa-analyzer` (2021), accessed: 2021-04

[20] Kondoh, G., Onodera, T.: Finding bugs in Java native interface programs. In: ISSTA, ACM (2008)

[21] Kristensen, E.K., Møller, A.: Reasonably-most-general clients for JavaScript library analysis. In: ICSE, IEEE / ACM (2019), URL `https://doi.org/10.1109/ICSE.2019.00026`

[22] Lee, S., Lee, H., Ryu, S.: Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis. In: ASE, IEEE (2020)

[23] Li, S., Tan, G.: Finding bugs in exceptional situations of JNI programs. In: CCS, ACM (2009)

[24] Li, S., Tan, G.: JET: exception checking in the Java native interface. In: SPLASH, ACM (2011)

[25] Li, S., Tan, G.: Exception analysis in the Java Native Interface. Sci. Comput. Program. (2014)

[26] Li, S., Tan, G.: Finding reference-counting errors in Python/C programs with affine analysis. In: ECOOP, Springer (2014)

[27] Malcolm, D.: A static analysis tool for CPython extension code. `https://gcc-python-plugin.readthedocs.io/en/latest/cpychecker.html` (2018), accessed: 2021-04

[28] Mao, J., Chen, Y., Xiao, Q., Shi, Y.: RID: finding reference count bugs with inconsistent path pair checking. In: ASPLOS, ACM (2016)

[29] Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. ACM Trans. Program. Lang. Syst. (3) (2009)

[30] Meyer, M.: Distance library. `https://github.com/doukremt/distance` (2021), accessed: 2021-04

[31] Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES, ACM (2006)

[32] Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006)

[33] Monat, R., Ouadjaout, A., Miné, A.: Static type analysis by abstract interpretation of Python programs. In: ECOOP, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)

[34] Monat, R., Ouadjaout, A., Miné, A.: Value and allocation sensitivity in static Python analyses. In: SOAP@PLDI, ACM (2020)

[35] Monat, R., Ouadjaout, A., Miné, A.: A Multi-Language Static Analysis of Python Programs with Native C Extensions (Jul 2021), URL `https://doi.org/10.5281/zenodo.5141314`

[36] Muła, W., Ombredanne, P.: Pyahocorasick library. `https://github.com/WojciechMula/pyahocorasick` (2021), accessed: 2021-04

[37] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: SAS, Springer (2020)

[38] Rinetzky, N., Poetzsch-Heffter, A., Ramalingam, G., Sagiv, M., Yahav, E.: Modular shape analysis for dynamically encapsulated programs. In: ESOP, Springer (2007)

[39] van Rossum, G., Lehtosalo, J., Łukasz Langa: Python Enhancement Proposal 484. `https://www.python.org/dev/peps/pep-0484/` (2021), accessed: 2021-08-03

[40] van Rossum, G., the Python development team: Python/C API Reference Manual. `https://docs.python.org/3.8/c-api/index.html` (2021), accessed: 2021-04

[41] Schnell, I.: Bitarray library. `https://github.com/ilanschnell/bitarray` (2021), accessed: 2021-04

[42] Tan, G., Croft, J.: An empirical security study of the native code in the JDK. In: USENIX, USENIX Association (2008)

[43] Tan, G., Morrisett, G.: Ilea: inter-language analysis across Java and C. In: OOPSLA, ACM (2007)

[44] Typeshed contributors: Typeshed. `https://github.com/python/typeshed/` (2021), accessed: 2021-04

[45] Wei, F., Lin, X., Ou, X., Chen, T., Zhang, X.: JN-SAF: precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code. In: SIGSAC, ACM (2018)