Towards Cross-Language Verification between Python and C

Etienne Birling and Marco Eilers(✉)

Department of Computer Science, ETH Zurich, Switzerland
etienne.birling@gmail.com, marco.eilers@inf.ethz.ch

Abstract. In many modern software systems, programs are written in multiple languages to balance performance and productivity. For the CPython runtime, modules are frequently implemented in C to enhance execution speed or interface with existing libraries. To ensure the correctness of such modules, developers must not only implement the intended functionality and maintain memory safety but also adhere to API rules and preserve runtime invariants.
In this paper, we present the first cross-language deductive verification technique for Python modules written in C. We define high-level semantic specification constructs and use them to bridge the abstraction gap between the two languages, translating Python contracts for the Nagini verifier into corresponding C contracts for VeriFast. We further provide a formal specification of a subset of the Python C API to ensure safety and the preservation of critical invariants, such as correct reference counting. We evaluate our implementation through a case study, demonstrating that our approach can successfully verify realistic "glue code" used in multi-language applications.

Keywords: Deductive verification, Python, foreign function interface, cross-language verification, separation logic

1 Introduction

In recent decades, formal methods have achieved significant milestones in verifying the correctness of real-world software. Despite this progress, complete system reliability remains elusive; even in fully verified systems, bugs can occur at component interfaces [11]. A particularly critical and complex interface is the *language boundary* [24], where high-level code invokes native functions through a Foreign Function Interface (FFI), such as the Java Native Interface (JNI) [17] or the CPython C API [12].

In the Python ecosystem, this boundary is especially crucial: the popularity of Python stems in large part from its ability to provide high-level, ergonomic access to performance-critical native code. CPython allows developers to implement modules in C that are indistinguishable from native Python modules to the end user. However, this seamless integration masks a stark architectural divide: Python is a high-level, managed language where everything is an object

and memory is managed automatically via reference counting, whereas C is a low-level language relying on manual memory management and direct pointer access. Consequently, when formally verifying Python programs that contain such modules, there is a potential *verification gap*: a program is only truly verified if correctness proofs span both the high-level Python logic and the low-level C implementation, including the shared memory and state between them.

To verify such programs, an obvious approach is to employ state-of-the-art verifiers for each programming language independently, such as Nagini [7] for Python and VeriFast [15], VerCors [3], SecC [8], VCC [6], or Frama-C [16] for C. However, ensuring correctness at the language boundary requires more than verifying each side in isolation. In particular, C code that implements Python modules must satisfy a broad spectrum of requirements. First, it must adhere to standard C safety properties, such as memory safety and the absence of undefined behavior. Second, it must comply with the rules imposed by the CPython C API, including correct handling of dynamically-typed values, proper error propagation, and adherence to rules for converting between Python and C representations. Third, it must preserve API-specific invariants, most notably the correct management of reference counts. Beyond these low-level concerns, the C implementation must also respect the (often implicit) invariants assumed by the Python-level verifier; for example, Nagini enforces that object fields contain only values of their annotated types, a property not guaranteed by the Python runtime itself. Finally, the C code must implement the intended functional behavior, producing results that satisfy some high-level specification.

This interplay of requirements gives rise to a fundamental specification challenge. On the one hand, when verifying Python code, the behavior of native functions must be captured by precise Python-level contracts, such as postconditions. On the other hand, verifying the corresponding C implementation requires assumptions about the conditions established by the Python caller (e.g., a call to an API function that extracts a long value from a Python object may require that the caller passed an integer-typed object whose value is in a specific range as an argument). However, such assumptions can only be expressed in terms of Python-level preconditions. Consequently, a sound verification approach must (1) express all relevant proof obligations across both languages and (2) provide a principled mechanism to relate these obligations across the language boundary.

Bridging this gap introduces several non-trivial scientific challenges:

- **Logic Heterogeneity**: Different verifiers rely on distinct underlying logics (e.g., implicit dynamic frames [23] in Nagini versus separation logic [20] in VeriFast), complicating the integration of proofs across tools.
- **Specification Mismatch**: Annotation languages differ significantly in syntax and semantics, making direct translation of specifications non-trivial.
- **Abstraction Gap**: C code operates at a much lower level of abstraction, exposing implementation details—such as memory layouts of objects or internal data structures—that are not visible at the Python level.

In this paper, we present the first framework for cross-language deductive verification between Python and C, specifically targeting C-implemented Python

functions exposed through the CPython API. We focus on the verification of such functions rather than classes or full modules, allowing us to isolate and address the core challenges of the language boundary. We provide a comprehensive analysis of these challenges and propose both high-level architectural strategies and principled concrete solutions.

Our framework is implemented using Nagini for Python and VeriFast for C. We deliberately select these verifiers because they use closely-related program logics, which significantly simplifies the task of addressing logic heterogeneity. By exploiting a common fragment of separation logic supported by both systems, we establish a shared reasoning foundation that bridges the two tools.

Building on this foundation, we define a translation framework between Python and C specifications. The central idea is to express C-level specifications exclusively in terms of high-level Python concepts, thereby enabling a unidirectional translation from Python specifications to C contracts. This design avoids the need to lift low-level implementation details to the Python level. To ensure that low-level modifications do not violate high-level assumptions, we leverage separation logic permissions to control access to internal data structures.

Finally, we provide contracts that systematically enforce correct usage of the CPython C API and guarantee the preservation of implicit runtime invariants, such as reference count integrity.

By bridging these two distinct verification ecosystems, our approach enables the formal verification of the performance-critical "glue code" that underpins modern Python applications, moving closer to end-to-end system guarantees.

To summarize, we make the following contributions:

- A High-Level Strategy for Native Interfaces: We propose a generalized approach for verifying native language interfaces, addressing the abstraction gap between high-level managed languages and low-level native code.
- C-Level Mapping of Python Specifications: We show how high-level Python specification constructs can be represented and reasoned about at the C level in separation logic.
- Formalization of the Python C API: We provide a specification of a subset of the CPython C API within VeriFast. As part of this effort, we introduce a novel technique for modular verification of reference counting consistency, ensuring memory safety and preventing leaks at the boundary.
- Automated Contract Translation: We implement an automated pipeline that translates specifications from Nagini (Python) into corresponding VeriFast (C) contracts.
- Empirical Evaluation: We evaluate our framework on a case study, demonstrating the feasibility of verifying typical "glue code" in practice.

2 Background: Python Modules in C

Python's C API enables developers to implement Python functions in C while preserving the illusion of ordinary Python functions at the call site. We illustrate the key concepts using the example shown in Listing 1.1, which implements a simple max function in C.

```c
#include <Python.h>

static PyObject* py_max(PyObject* self, PyObject* args) {
    // Check if 'args' is a tuple with 2 elements
    if (!PyTuple_Check(args) || PyTuple_Size(args) != 2) {
        PyErr_SetString(PyExc_TypeError, "Expected exactly 2 arguments");
        return NULL;
    }

    // Get items from tuple
    // PyTuple_GetItem returns a borrowed reference
    PyObject* obj_a = PyTuple_GetItem(args, 0);
    PyObject* obj_b = PyTuple_GetItem(args, 1);

    // Convert to C longs
    long val_a = PyLong_AsLong(obj_a);
    // Return with exception if value out of range
    if (PyErr_Occurred()) return NULL;

    long val_b = PyLong_AsLong(obj_b);
    if (PyErr_Occurred()) return NULL;

    // Compare using standard C operators
    PyObject* winner;
    if (val_a > val_b) {
        winner = obj_a;
    } else {
        winner = obj_b;
    }

    // Since 'winner' is pointing to one of the borrowed objects from 'args',
    // we must INCREF it before returning so the caller owns it.
    Py_INCREF(winner);
    return winner;
}
```

Listing 1.1. Example function implemented in C.

Function structure and module integration. C functions exposed to Python via the CPython API have a standardized signature. In our example, the function py_max takes two arguments of type PyObject* and returns a PyObject*. The first parameter, conventionally named self, refers to the module or object the function is associated with and will not be used here, while the second parameter, args, encodes the actual arguments passed from Python. The return value is either a pointer to a Python object representing the result or NULL to indicate that an exception has occurred.

Argument passing via tuples. In the CPython calling convention used here, positional arguments are passed as a tuple bundled into the single parameter args. Thus, a Python call such as max(3, 4) results in args being a tuple of length two, containing two Python integer objects. The first step in the function is therefore to validate that args has the expected structure. This is achieved using the API functions PyTuple_Check and PyTuple_Size.

Once validated, individual arguments are accessed using PyTuple_GetItem, which retrieves elements by index. Importantly, this function returns a *borrowed*

reference, meaning that the caller does not obtain ownership of the returned object. Instead, the lifetime of the object remains tied to the original tuple.

Extracting values and handling exceptions. After retrieving the tuple elements, the function converts them to native C values using PyLong_AsLong, which extracts a C long from a Python integer object. However, this operation may fail if the object is not an integer or if its value exceeds the representable range of a C long. Rather than signaling errors through return values alone, the CPython API uses a global exception indicator that is set when an error occurs.

To detect such failures, the code explicitly checks PyErr_Occurred() after each conversion. If an exception is present, the function returns NULL, propagating the error back to the Python caller. This pattern of calling an API function, checking for an exception, and aborting if necessary is idiomatic in C extensions.

Reference counting and ownership. Memory management in CPython is based on reference counting. Each PyObject maintains a count of how many references to it exist, and the object is deallocated when this count drops to zero. The C API enforces a disciplined ownership model: functions either return *new references*, meaning that the caller becomes responsible for decrementing the reference when it is no longer needed, or *borrowed references*, which must not be decremented by the caller.

In the example, PyTuple_GetItem returns borrowed references to the tuple elements. Consequently, the C code does not own these references and must not decrement their reference counts. However, when returning a value to the Python caller, the convention is that the caller receives a new reference. Therefore, before returning one of the tuple elements as the result, the function must increment its reference count using Py_INCREF. This step is essential: without it, the returned object would still be owned solely by the tuple args. Once the tuple is no longer needed, its reference count may drop, potentially leading to the deallocation of the object while it is still in use by the caller, i.e., a use-after-free. By incrementing the reference count, the function ensures that the returned object remains valid independently of the tuple's lifetime.

3 Specification Translation

In this section, we present our overall approach to cross-language verification and introduce its central technical component: a translation framework between Python contracts written for Nagini and C contracts verified with VeriFast.

The main difficulty at the language boundary is the *abstraction gap* between Python and C. Python programs and their specifications operate on high-level concepts such as integer and list objects, whereas C implementations work with PyObjects that internally contain actual representations of integers or lists. Directly reasoning about these low-level details at the Python level would both complicate specifications and undermine abstraction.

To address this challenge, we adopt a deliberately asymmetric design: we identify a set of core specification constructs that are meaningful at the Python

level and restrict all cross-language reasoning to these constructs. In particular, we avoid exposing low-level implementation details, such as the internal representation of PyObjects, in specifications. Instead, we express all relevant properties in terms of high-level, Python-visible behavior.

Building on this idea, we provide specifications for the Python C API itself using these high-level constructs. As a consequence, when C code invokes API functions, it must satisfy preconditions that are expressed in terms of Python-level abstractions. These preconditions can, in turn, be established by the calling Python code during its own verification. To ensure that C implementations cannot violate the intended abstraction boundaries, we leverage separation logic permissions to prevent direct modifications of internal data structures.

Once this common specification layer is established, we define a *top-down* translation from Python contracts to C contracts. Concretely, Python functions are specified in Nagini using preconditions and postconditions. These specifications are then automatically translated into corresponding VeriFast contracts attached to the C function implementations. This yields a streamlined workflow: the developer of a Python C extension first writes a stub file containing type signatures and Nagini specifications, then translates these into C-level function signatures and VeriFast contracts, and finally implements and verifies the C code against the generated contracts.

An additional technical challenge in this process is the mismatch between the underlying program logics: Nagini is based on implicit dynamic frames [23], whereas VeriFast uses separation logic [20]. We deliberately chose these two verifiers because their logics are closely related and share a native focus on ownership, which significantly simplifies the translation. Instead of attempting a full mapping between the systems, we restrict ourselves to a shared subset that includes separation logic predicates, pure mathematical functions, and ownership based on fractional permissions [4].

Even between these relatively similar frameworks, a substantial part of our translation effort is devoted to reconciling their semantic differences, as discussed in Sec. 3.4. We believe that a translation between entirely disparate program logics would introduce immense additional complexity in addition to the challenges inherent to the Python-C interface itself.

The remainder of this section elaborates on these ideas. We begin by describing the general structure of the generated verification setup, including function signatures and argument handling conventions. We then introduce the core specification constructs that form the backbone of our approach and show how they can be used to specify the behavior of Python C API functions. Finally, we present the details of our contract translation from Python to C, highlighting the key design decisions and their implications for verification.

3.1 Setup

We start from a Python function annotated with Nagini specifications, as shown in Listing 1.2. The @Native decorator indicates that the function is implemented in C rather than Python, and thus requires cross-language verification.

```
@Native
def foo(arg_1: T_1, ..., arg_n: T_n) -> T_r:
    Requires(P)
    Ensures(Q)
    Exsures(Ex_1, Q_Ex_1)
```

Listing 1.2. Python function stub with specification, from which we generate a C function specification.

```
@Native
def max(a: int, b: int) -> int:
    Requires(LONG_MIN < a < LONG_MAX and LONG_MIN < b < LONG_MAX)
    Ensures(Result() is a if a > b else b)
```

Listing 1.3. Python-level specification for the max function implemented in Listing 1.1.

We assume that all functions are fully annotated with types following the standard PEP 484 [21]. The specification consists of standard pre- and post-conditions written using Requires and Ensures, respectively. In addition, exceptional behavior can be specified using Exsures clauses. An annotation of the form Exsures(Ex_1,Q_{Ex_1}) states that the function may raise an exception of type Ex_1, and that in this case the exceptional postcondition Q_{Ex_1} must hold instead of the normal postcondition. Any exception that is not explicitly listed in an Exsures clause is disallowed.

For the max function from Listing 1.1, a corresponding specification is shown in Listing 1.3. The precondition ensures that both arguments lie within the range representable as C long values, which guarantees that the conversion via the C API will not fail. The postcondition states that the function returns the greater of the two inputs, and no exceptional behavior is specified, implying that the function must not raise any exceptions.

From such a Python specification, we automatically generate a corresponding C function signature with a VeriFast contract, whose structure is illustrated in Listing 1.4.

The generated contract captures several aspects of the execution model. First, it encodes the structure of the argument tuple and binds both the low-level PyObject* pointers and their corresponding high-level values. Second, it embeds the translated Python precondition $[\![P]\!]_{\sigma_0}$ into the C-level precondition. Third, it models the global exception state and enforces that only declared exceptions may be raised. Finally, it maintains a module invariant Inv that captures consistency properties of the Python module implemented in C.

We will explain each component of this generated specification in more detail in the remainder of this section and the next. Note that reference counting, which involves the hasRef predicates shown throughout Listing 1.4, as well as the module invariant Inv will be discussed in Sec. 4.

```
static PyObject* foo(PyObject* self, PyObject* args)
    requires hasRef(args, false) &*&  // reference counting
             // value binding for arguments
             hasVal(args, VTuple([(?arg_1_p, T_1), ..., (?arg_n_p, T_n)] &*&
             hasVal(arg_1_p, val(arg_1_v, T_1)>) &*&
             ... &*&
             hasVal(arg_n_p, val(arg_n_v, T_n)>) &*&
             // encoded Python precondition
             [[P]]_σ0 &*&
             // exception state
             PyExc(none, none) &*&
             // module invariant
             Inv
    ensures  // reference counting
             hasRef(args, false) &*&
             hasRef(result, true) &*&
             // value binding for result
             hasVal(result, val(res_v, T_r)>) &*&
             // exception state
             PyExc(?e_new, ?t_new) &*&
             (t_new == none) ?
                 ([[Q]]_σ0) : // regular postcondition
                 (t_new == some(ex1) ?
                     [[Q_Ex1]]_σ0 : // exceptional postcondition
                     false) &*&
             Inv
```

Listing 1.4. C function and specification generated from the Python stub in Listing 1.2.

3.2 Specification Constructs

Pointers and Values: A central challenge in our approach is to relate low-level PyObject* pointers to the high-level values they represent at the Python level. For instance, we must be able to refer to the integer value stored in a Python integer object, or to the sequence of elements stored in a tuple object. To express this relation, we introduce the abstract VeriFast predicate hasVal, shown in Listing 1.5.

The predicate hasVal(pyobj, v) connects a PyObject to an abstract representation of its value. For primitive types, such as integers and booleans, this value directly corresponds to the underlying mathematical value (e.g., VLong). For

```
inductive PyClass =
    PyClass ( list<char>, PyClass, list<PyObj_Type>) |
    ObjectClass;

inductive PyObj_Val =
    VBool(bool) |
    VLong(int) |
    VClassInstance(PyClass) |
    VTuple(list< pair<PyObject *, PyObj_Type> >) |
    ...;

predicate hasVal(PyObject* pyobj; PyObj_Val v);
```

Listing 1.5. Predicate relating PyObjects to the values they contain.

```
predicate PyExc_Flag(option<PyObject*>);
predicate PyExc(option<PyObject*> e, option<PyClass> type) =
    PyExc_Flag(e) &*&
    switch (e) {
        case none: return type == none;
        case some(x): return type==some(?y) &*& hasVal(x, VClassInstance(y));
    };
predicate hasAttr(PyObject* o, list<char> n; PyObject* out);
```

Listing 1.6. Predicates that specify the current contents of Python's internal exception flag and its type, as well as the value stored in an object attribute.

compound objects such as tuples, the value captures their structure in terms of contained objects and their types. For general class instances, the value records only type information.

Crucially, the value associated with a PyObject contains only *immutable* information. Mutable aspects of objects are handled separately through permissions, ensuring that values remain stable across program steps.

In the generated specification (Listing 1.4), the hasVal predicate is used to describe the structure of the args tuple and to bind names to both argument pointers and their corresponding values. We make use of VeriFast's pattern matching notation: an expression of the form ?id matches an arbitrary value and binds it to the identifier id. For example, the tuple pattern introduces the name arg_i_p for the PyObject* pointer corresponding to each argument i. Subsequently, additional hasVal predicates bind name arg_i_v to the abstract value of this object.

The notation $val(\text{arg\_i\_v}, T_i)$ abbreviates the construction of a value consistent with type T_i and binds the value to the new name arg_i_v. For instance, if T_i is int, this expands to VLong(?arg_i_v). This separation between pointers and values allows specifications to refer naturally to high-level data while maintaining a precise link to the underlying C objects.

For our max function, both argument types are int, and thus we get two predicates hasVal(arg_i_p, VLong(?arg_i_v)); now, we can refer to the *pointer* corresponding to argument i as arg_i_p, and to the *integer value* of that argument as arg_i_v.

Exceptions: A second core construct is the predicate PyExc, defined in Listing 1.6, which models Python's global exception state.

The predicate PyExc(e, type) captures both the currently raised exception object (if any) and its type. In the precondition of Listing 1.4, we require that no exception is currently active. In the postcondition, we allow either normal termination (no exception) or exceptional termination, provided that the exception type matches one of the declared Exsures clauses. This encoding directly connects the C-level error signaling mechanism to the Python-level specification of exceptional behavior.

Attributes: Finally, we introduce the predicate hasAttr, also shown in Listing 1.6, for reasoning about object attributes. Conceptually, hasAttr(o, "f", v) is anal-

```
PyObject* PyObject_GetAttrString(PyObject *o, const char *attr_name);
    requires hasRef(o, ?own) &*&
             string(attr_name, ?attr_name_val) &*&
             [?f]hasAttr(o, attr_name_val, ?o_attr_ptr);
    ensures  hasRef(o, own) &*&
             string(attr_name, attr_name_val) &*&
             [f]hasAttr(o, attr_name_val, o_attr_ptr) &*&
             hasRef(result, true) &*&
             result==o_attr_ptr;

int PyObject_SetAttrString(PyObject *o, const char *attr_name, PyObject *v);
    requires hasRef(o, ?ownO) &*& hasRef(v, ?ownV) &*&
             string(attr_name, ?attr_name_val) &*&
             hasAttr(o, attr_name_val, ?o_attr_ptr);
    ensures  hasRef(o, ownO) &*& hasRef(v, ownV) &*&
             string(attr_name, attr_name_val) &*&
             hasAttr(o, attr_name_val, v) &*&
             result == 0;
```

Listing 1.7. Functions for reading and modifying object attributes, specied using hasAttr. Note that reading an attribute only requires some fractional amount f of the predicate, whereas writing it requires a full permission.

ogous to a standard separation logic points-to assertion: it grants permission to access the attribute f of object o, whose current value is v. Unlike hasVal, which captures immutable, high-level properties, hasAttr governs mutable state and enforces controlled access through permissions. We use a similar predicate to model the contents of mutable *containers* like lists, sets and dictionaries, but will not discuss it here, since the basic idea is the same.

Together, the constructs hasVal, PyExc, and hasAttr form the core vocabulary for expressing specifications that bridge the Python and C levels while preserving a clean separation between abstraction layers.

3.3 API Specification

We use the specification constructs introduced in the previous subsection to formally describe the behavior of functions in the Python C API. The key idea is that API functions are specified entirely in terms of high-level predicates such as hasVal, hasAttr, and PyExc, thereby avoiding any dependence on low-level implementation details. This ensures that both the specifications of API functions and the verification of client code operate on the same abstraction layer as Python-level reasoning. Listing 1.7 illustrates this approach for two representative API functions that read and modify object attributes. Their behavior is specified using the hasAttr predicate.

The specification of PyObject_GetAttrString expresses that reading an attribute requires only fractional permission f to the corresponding hasAttr predicate. This reflects the fact that attribute access is a read-only operation that does not modify the object state.

In contrast, PyObject_SetAttrString requires full permission to the attribute, indicating exclusive access necessary for mutation. The postcondition reflects

```
PyObject* PyLong_FromLong(long x);
    requires true;
    ensures  hasVal(result, PyLong_v(x)) &*& hasRef(result, true) &*&
             result != NULL;

long PyLong_AsLong(PyObject *p);
    requires PyExc(?e, ?t) &*& hasRef(p, ?o) &*& hasVal(p, PyLong_v(?v));
    ensures  PyExc(?e_new, ?t_new) &*&
             hasRef(p, o) &*& hasVal(p, PyLong_v(v)) &*&
             ((v <= LONG_MAX && v >= LONG_MIN) ?
                 (e == e_new &*& t == t_new &*& result == v) :
                 (result == -1 &*& e != e_new &*& e_new == some(_) &*&
                  t_new == some(OverflowError)));

PyObject* PyErr_Occurred();
    requires PyExc(?ptr, ?type);
    ensures  PyExc(ptr, type) &*&
             switch (ptr) {
               case none:
                 return result == NULL;
               case some(x):
                 return result != NULL &*& result == x;
             };
```

Listing 1.8. Functions for converting between Python int objects and C long values, specied using PyExc and hasVal predicates. Function PyErr_Occurred retrieves the current exception, if any.

that the attribute now points to the new value v. In both cases, the specifications enforce disciplined access to object state via separation logic permissions, ensuring that aliasing and mutation are properly controlled.

Listing 1.8 shows specifications for the API functions that convert between Python integer objects and C long values. These specifications demonstrate how hasVal and PyExc are used to relate high-level values and exception behavior.

The specification of PyLong_FromLong is straightforward: it constructs a fresh Python integer object representing the given C value and returns a new reference to it. This is captured by the hasVal predicate.

The function PyLong_AsLong is more subtle, as it may fail if the Python integer cannot be represented as a C long. Its specification distinguishes two cases. If the value lies within the valid range, the function returns the corresponding C value and leaves the exception state unchanged. Otherwise, it signals an error by returning -1 and updating the global exception state to an OverflowError. This behavior is expressed by explicitly relating the pre- and post-state of the PyExc predicate.

Function PyErr_Occurred allows checking the current exception state and returns a pointer to the current exception object, if any.

These examples illustrate how API specifications expose only the high-level semantics required for verification while abstracting away from low-level implementation details. As we show in the next subsection, these specifications form the foundation for translating Python-level contracts into corresponding C-level verification conditions.

3.4 Translation

Using the specification constructs introduced above, we now define a translation from (a subset of all) Nagini specifications to VeriFast specifications. This translation forms the core of our approach, as it connects high-level Python contracts to low-level C verification conditions in a systematic way.

Two main challenges arise in this process. First, a Python expression may refer both to its *value* (e.g., `x > 0`) and to its *reference identity* (e.g., `x is y`). In contrast, VeriFast specification have to distinguish between pointers and the abstract values they represent. The translation must therefore determine, for each expression, whether its pointer-level or value-level representation is required.

Second, due to the use of different program logics, there is a mismatch in how heap-dependent expressions are handled. In Nagini, specifications can freely refer to heap locations and their contents within a single assertion, for example `Acc(x.f) and x.f > 0`[1]. In VeriFast, accessing a heap location requires binding the referenced object and its value to fresh identifiers. Subsequent uses of the attribute must use these identifiers rather than accessing the heap directly (e.g., `hasAttr(x, "f", ?x_f_p) &*& hasVal(x_f_p, VLong(?v)) &*& v > 0`).

To address these challenges, we introduce a *translation context* σ that records how parameters and heap-dependent expressions are mapped to VeriFast identifiers. For each expression, the context stores two entries: one for its pointer representation and one for its value. The initial context σ_0 contains entries for all function parameters, such as `arg_i_p` and `arg_i_v` in Listing 1.4. As the translation proceeds, additional entries are added whenever new attribute permissions are encountered. For example, when translating an attribute permission `Acc(x.f)`, the context is extended with fresh identifiers (e.g., `x_f_p` and `x_f_v`) representing the attribute's pointer and value, respectively. Later references to an argument or a heap location look them up in the context. Note that due to aliasing, this translation process is incomplete, as lookups in the context for semantically equivalent but syntactically different expressions can fail; however, it is always possible to rewrite specifications to a supported form.

Based on this context, we define a translation function $[\![P]\!]^m_\sigma$, which maps a Nagini assertion or expression P to a pair $\langle P', \sigma' \rangle$ consisting of a VeriFast assertion or expression P' and an updated context σ'. The parameter $m \in p, v$ indicates whether the translation should yield the pointer-level (p) or value-level (v) representation of an expression.

Figure 1 shows an excerpt of this translation function for relevant cases. Conjunctions are translated compositionally, threading the context through both sub-expressions. Heap permissions expressed via `Acc`(e.`f`) are translated into `hasAttr` predicates, while simultaneously introducing fresh identifiers for the attribute's pointer and value and recording them in the context.

Conditional assertions, expressed using `Implies`, require special care: since the condition may not hold at runtime, any bindings introduced when translating

[1] Note that a reference to `x.f` in a specification that does not also contain the relevant permission `Acc(x.f)` is not well-defined; we assume that all specifications we translate are well-defined.

$$
\begin{aligned}
[\![P \text{ and } Q]\!]_\sigma &= \langle P' \ \&*\& \ Q', \sigma'' \rangle \\
&\quad \text{where } \langle P', \sigma' \rangle = [\![P]\!]_\sigma \text{ and } \langle Q', \sigma'' \rangle = [\![Q]\!]_{\sigma'} \\
[\![\mathsf{Acc}(e.\mathsf{f})]\!]_\sigma &= \langle \mathsf{hasAttr}([\![e]\!]_\sigma, \texttt{"f"}, \ \texttt{?e\_f\_ptr}) \ \&*\& \\
&\quad \mathsf{hasVal}(\mathsf{e\_f\_ptr}, \mathit{val}(\mathsf{e\_f\_val}, T)), \sigma' \rangle \\
&\quad \text{where } \sigma' = \sigma[\langle e.f, p \rangle \mapsto \mathsf{e\_f\_ptr}, \langle e.f, v \rangle \mapsto \mathsf{e\_f\_val}] \\
&\quad \text{and } T \text{ is the type of field f} \\
[\![\mathsf{Implies}(e, P)]\!]_\sigma &= \langle e' \ ? \ P' \ : \ \mathsf{true}, \sigma \rangle \\
&\quad \text{where } \langle e', \_ \rangle = [\![e]\!]^v_\sigma \text{ and } \langle P', \_ \rangle = [\![P]\!]_\sigma \\
[\![x]\!]^p_\sigma &= \sigma[\langle \mathsf{x}, p \rangle] \\
[\![x]\!]^v_\sigma &= \sigma[\langle \mathsf{x}, v \rangle] \\
[\![e.f]\!]^p_\sigma &= \sigma[\langle \mathsf{e.f}, p \rangle] \\
[\![e.f]\!]^v_\sigma &= \sigma[\langle \mathsf{e.f}, v \rangle] \\
[\![e_1 < e_2]\!]_\sigma &= e'_1 < e'_2 \\
&\quad \text{where } \langle e'_1, \_ \rangle = [\![e_1]\!]^v_\sigma \text{ and } \langle e'_2, \_ \rangle = [\![e_2]\!]^v_\sigma \\
[\![e_1 \text{ is } e_2]\!]_\sigma &= e'_1 == e'_2 \\
&\quad \text{where } \langle e'_1, \_ \rangle = [\![e_1]\!]^p_\sigma \text{ and } \langle e'_2, \_ \rangle = [\![e_2]\!]^p_\sigma
\end{aligned}
$$

Fig. 1. Excerpt of the definition of specification translation. Acc(e.f) expresses a permission to field f of object e in Nagini and is translated to a hasAttr predicate. The corresponding pointer e_f_ptr and value e_f_ptr are stored in the translation context σ. When translating assertions under a condition (in the Implies case), the context is not updated, since names defined under that condition will not be available if the condition is false and therefore cannot be used outside that branch.

the right hand side cannot safely be used outside the conditional. Therefore, the translation of Implies does not propagate context updates from its body.

The translation also makes explicit the distinction between pointer and value equality. A comparison such as $\mathsf{e}_1 < \mathsf{e}_2$ is translated by evaluating both operands at the value level, whereas identity comparisons using is are translated to pointer equality in VeriFast.

The result $[\![P]\!]_{\sigma_0}$ of translating the Nagini precondition P shown in Listing 1.3 is the VeriFast assertion LONG_MIN < arg_1_v < LONG_MAX &*& LONG_MIN < arg_2_v < LONG_MAX; the translated postcondition $[\![Q]\!]_{\sigma_0}$ is result == (arg_1_v > arg_2_v ? arg_1_p : arg_2_p).

4 Maintaining Invariants

The previous section introduced our approach to expressing explicit specifications for both user-defined C functions and Python C API functions. However, functional correctness alone is not sufficient: correct interaction with the Python runtime also requires maintaining a number of implicit invariants that are not directly visible in high-level specifications.

In particular, C implementations must preserve several classes of invariants: (i) *runtime invariants* required by CPython, most notably correct reference counting, (ii) *verifier invariants* assumed by Nagini (e.g., type consistency of object attributes), and (iii) *module-specific invariants* that arise from the internal design of a given C module.

In the following subsections, we discuss how these invariants are captured and enforced within our verification framework.

4.1 Runtime Invariant: Reference Counting

As discussed in Section 2, CPython manages memory using reference counting. Each object maintains a count of how many references to it exist, and objects are deallocated once this count reaches zero. The C API distinguishes between *owned* (new) references and *borrowed* references. Owned references contribute to the reference count and must eventually be released by the caller, whereas borrowed references are temporarily usable but must not be decremented.

C code can explicitly manipulate reference counts using Py_INCREF and Py_DECREF. Incrementing the reference count creates a new owned reference, while decrementing consumes one owned reference and may trigger deallocation if the count reaches zero.

From a verification perspective, reference counting poses a significant challenge. In general, the absolute reference count of an object is unknown: a function may receive arguments that are referenced elsewhere in the program, and objects reachable through fields or container structures may have arbitrarily many aliases. Therefore, it is not feasible to reason about exact reference counts. Instead, the goal is to prevent use-after-free errors and memory leaks and ensure that reference ownership is respected, without requiring knowledge of global reference counts.

To address this, we introduce the predicate hasRef(o, owned), shown in Listing 1.9, which models the possession of a single reference to object o. The boolean flag owned indicates whether this reference is owned by the current context or borrowed from elsewhere.

Intuitively, hasRef(o, owned) represents the right to use the object o, guaranteeing that it has not been deallocated. Unlike hasVal, for hasRef, the distinction between owning one or multiple predicate instances for the same PyObject is significant, as each instance represents one held reference. The distinction between owned and borrowed references enforces correct usage of the API: Py_INCREF requires any reference (owned or borrowed) to ensure the object is still alive, and

```
predicate hasRef(PyObject *o, bool owned);

void Py_INCREF(PyObject *obj);
    requires hasRef(obj, ?o);
    ensures  hasRef(obj, o) &*& hasRef(obj, true);

void Py_DECREF(PyObject *obj);
    requires hasRef(obj, true);
    ensures  true;
```

Listing 1.9. Predicate for modelling relative reference count (the flag owned models whether a given reference is owned or borrowed), and specifications for INCREF and DECREF.

```
int PyList_SetItem(PyObject *list, Py_ssize_t index, PyObject *item);
    requires hasRef(list, ?o) &*& hasRef(item, true) &*&
             hasVal(list, PyList_v(?t)) &*&
             0 <= index &*& index < length(c) &*& ...;
    ensures  hasRef(list, o) &*& hasVal(list, PyList_v(t)) &*&
             ...;
```

Listing 1.10. PyList_SetItem is an example of an API function that *steals* a reference: It requires an owned reference to the item and does not return it to the caller.

produces a new owned reference; Py_DECREF consumes an owned reference and is therefore only permitted when such ownership is available.

All interactions with PyObjects require possession of a corresponding hasRef predicate. This is reflected in the specifications of previously shown API functions such as PyLong_AsLong and PyObject_GetAttrString, which require a reference (i.e., an instance of hasRef) for each of their arguments. Functions that return new references, such as PyLong_FromLong and PyObject_GetAttrString, provide ownership to the caller via hasRef(result , true) in their postconditions.

Some API functions follow a different ownership discipline. Listing 1.10 shows the specification of PyList_SetItem, a function that *steals* a reference: it consumes an owned reference passed as an argument and does not return it to the caller.

Handling borrowed references is more complex. In the generated function contract (Listing 1.4), the argument tuple is provided as a borrowed reference, which must be preserved across the function call. However, the elements of the tuple are also accessed via borrowed references. To safely use these elements, we must ensure that the tuple itself remains alive for the duration of their use.

To support this pattern, we introduce ghost operations for temporarily borrowing references from tuples, shown in Listing 1.11. The function borrowRefs consumes the tuple reference and produces borrowed references to each element (expressed as an iterated separating conjunction using bigstar). Conversely, returnRefs reassembles these references and restores the original tuple reference. The predicate borrowedTupleRefs represents the right to exchange references to all objects contained in a tuple back for a reference to the tuple itself.

Notably, PyTuple_GetItem itself does not produce a hasRef predicate for the returned object; it merely exposes the pointer stored in the tuple. To actually

```
predicate borrowedTupleRefs(PyObject *t,
                            list<pair<PyObject *, PyObj_Type> >,
                            bool owned);

predicate_ctor hasBorrowedRef()(PyObject *o) =
  hasRef(o, false);

void borrowRefs(PyObject *t);
    requires hasVal(t, PyTuple_v(?vls)) &*& hasRef(t, ?o);
    ensures  hasVal(t, PyTuple_v(vls)) &*&
             borrowedTupleRefs(t, vls, o) &*&
             bigstar(hasBorrowedRef(), map(fst, vls));

void returnRefs(PyObject *t);
    requires hasVal(t, PyTuple_v(?vls)) &*&
             borrowedTupleRefs(t, vls, ?o) &*&
             bigstar(hasBorrowedRef(), map(fst, vls));
    ensures  hasVal(t, PyTuple_v(vls)) &*& hasRef(t, o);

PyObject* PyTuple_GetItem(PyObject *p, Py_ssize_t pos);
    requires hasRef(p, ?o) &*& hasVal(p, PyTuple_v(?lst)) &*&
             0 <= pos &*& pos < length(lst);
    ensures  hasRef(p, o) &*& hasVal(p, PyTuple_v(lst))  &*&
             result==fst(nth(pos, lst));
```

Listing 1.11. Ghost functions for borrowing references from tuples: borrowRefs allows us to extract the references held by a tuple temporarily (in the form of an iterated separating conjunction, modelled using the bigstar predicate), temporarily consuming the tuple reference to prevent deallocating it, and returnRefs returns those references to the tuple and gives us back the original reference to the tuple.

use the returned object, the caller must first invoke borrowRefs to obtain the corresponding borrowed references. This design enforces that borrowed references can be used only while the containing tuple is guaranteed to remain alive.

In addition to preventing use-after-free errors, our specification discipline also ensures that reference leaks cannot occur. In particular, the generated function contracts require that the implementation returns *only* the borrowed references it received (e.g., the reference to the argument tuple) and the explicitly specified owned references (namely, the return value). Since VeriFast does not permit dropping resources silently, any additional hasRef instances that remain owned at the end of the function would constitute a verification error. As a result, correct reference count management (including the absence of leaks) is enforced automatically by the verification process.

Overall, this approach enables precise reasoning about reference ownership without requiring knowledge of global reference counts. It ensures that all object accesses are safe, that owned references are properly managed, and that borrowed references are used only within their valid lifetime.

4.2 Verifier Invariants

In addition to runtime invariants, C implementations must also preserve invariants imposed by the Python verifier. These invariants are not required for the correct execution of arbitrary Python programs, but they are guaranteed to hold

for verified Python code and are relied upon during verification. Consequently, any C code that interacts with verified Python components must maintain these invariants as well.

Two representative examples highlight the nature of such verifier invariants. First, Nagini enforces that object attributes only contain values consistent with their declared types. While Python itself does not enforce such constraints at runtime, verified code may rely on them. At the C level, however, nothing prevents assigning an object of an incompatible type to a field, for instance, storing a string object in an attribute that is expected to hold an integer.

Second, Nagini assumes that certain data structures, such as tuples, are immutable. Although this assumption holds at the Python level, the C API exposes the function PyTuple_SetItem that can modify tuple contents directly, thereby violating this invariant.

We address the first issue by strengthening the specification of attribute permissions. As shown in Fig. 1, the translation of Acc(e.f) introduces not only a hasAttr predicate, but also a corresponding hasVal constraint that encodes the expected type of the attribute. As a result, any attribute that is accessible to the C code and subsequently returned to the Python context must satisfy the appropriate type constraint in the postcondition. This ensures that type consistency is preserved across the language boundary.

The second issue is handled more conservatively. Since tuple immutability is a fundamental assumption in Nagini, we prohibit any operation that could violate it. Concretely, we assign the function PyTuple_SetItem a precondition of false, effectively disallowing its use in verified C code. This design choice ensures that the immutability invariant is preserved without requiring additional reasoning about low-level tuple manipulation.

It is worth noting that verifier invariants are not unidirectional: C-level verification can also rely on guarantees established by the Python verifier. For example, decrementing a reference count via Py_DECREF may trigger object deallocation, which in turn may invoke a __del__ method. In principle, such methods could have arbitrary side effects, complicating verification. However, Nagini enforces that __del__ methods are free of observable side effects. This guarantee allows us to treat deallocation as side effect free in VeriFast, thereby simplifying the reasoning about reference counting.

4.3 Module Invariants

The third class of invariants arises from the internal design of the C module itself. In addition to interacting with Python objects, a module may maintain its own state across function calls, including references to Python objects that persist beyond the scope of a single function invocation.

To ensure correctness, such state must satisfy a module-specific invariant that is preserved by all exported functions. In particular, if the module stores references to Python objects, it must maintain corresponding hasRef predicates to ensure that these objects are not deallocated prematurely. These ownership

constraints, along with any additional consistency properties, are captured in a module invariant, denoted by Inv in the generated specification in Listing 1.4.

The module invariant is defined by the implementer of the C extension and is assumed in the precondition and re-established in the postcondition of every exported function. Additionally, it must be shown to hold initially during module initialization. This mechanism enables modular reasoning about persistent state, ensuring that all interactions with Python objects remain safe and consistent across function boundaries.

5 Implementation and Case Study

We have implemented our approach as a prototype toolchain that connects Nagini and VeriFast through the translation framework described in the previous sections. Concretely, we provide a formal specification of a subset of the Python C API using the specification constructs introduced earlier, including predicates for values, attributes, exceptions, and reference ownership. Our implementation and case studies are available online[2].

On the Python side, Nagini is extended to generate verification artifacts for all functions annotated with `@Native`. From such annotated stubs, our tool automatically produces corresponding C function skeletons together with VeriFast specifications, following the scheme outlined in Section 3. In addition to translating pre- and postconditions, the implementation also supports the translation of Nagini predicates and pure mathematical functions, provided they do not depend on the heap. This restriction aligns with the capabilities of VeriFast's separation logic, which does not support heap-dependent pure functions.

To validate the expressiveness and practicality of our framework, we conducted a series of smaller tests that exercise individual aspects of the API specification, such as reference management, attribute access, and exception handling, confirming that the specifications enforce correct usage patterns.

As a more substantial evaluation, we verified a case study based on GMPY2, a library for high-performance multiple-precision arithmetic[3]. GMPY2 provides Python bindings for the GMP (GNU Multiple Precision), MPFR (floating-point), and MPC (complex arithmetic) libraries. We focused on the function `GMPY_MPZ_Function_Bincoef`, which computes binomial coefficients.

For this case study, we wrote a functional specification at the Python level, expressing the binomial coefficient using a recursive definition. This specification was automatically translated into a corresponding VeriFast contract by Nagini. We then verified the C implementation of the function against this contract.

The verification required minor adaptations of the original GMPY2 code. First, GMPY2 occasionally uses internal helper functions instead of standard Python C API functions when manipulating `PyObject`s; these were either speci-

[2] `https://github.com/marcoeilers/nagini/blob/c_api_interaction/src/nagini_translation/native/README.md`

[3] `https://gmpy2.readthedocs.io/`

fied separately or replaced with equivalent API calls. Second, some adjustments were necessary to accommodate current limitations of VeriFast.

Finally, we treat the underlying numerical library (MPZ) as a trusted component. Its correctness is assumed via a functional specification that describes its behavior at a high level. This allows us to focus on verifying the correctness of the Python-C interface code, which is the primary target of our framework.

Overall, the case study demonstrates that our approach is capable of handling realistic, performance-critical "glue code", and that cross-language deductive verification between Python and C is feasible in practice.

6 Related Work

Cross-language program verification has been an active research topic for many years. Some existing works, like ours, aim to build on language-specific verification tools and make them interoperable. Armborst et al. [1] translate specifications between different Java verifiers, while more general approaches provide intermediate specification languages [5]. ContractLIB [9] is a recent proposal for a universal contract language designed to express behavioral aspects of programs while abstracting over concrete program logics. While our current implementation uses an ad-hoc translation between the logic-agnostic parts of Nagini's and VeriFast's specification languages, much of this could, in principle, be automated using future ContractLIB translations for both tools.

In addition to intermediate specification languages, intermediate verification languages (IVLs) such as Viper [19], Boogie [2], and Why3 [10] are commonly used by verification tools and can address specific subproblems of inter-language verification. While IVLs do not reconcile differing source language semantics, two languages encoded into the same IVL necessarily share its underlying mathematical tools and program logic. For instance, had we targeted VerCors instead of VeriFast for C verification (which, like Nagini, is based on Viper) we would not have encountered the challenges stemming from mismatched program logics (we nevertheless chose VeriFast due to better support for some relevant C language features).

While the aforementioned approaches focus on connecting verification tools in practice, theoretical frameworks also exist for formally defining the semantics of multi-language programs and foundationally proving their properties [22].

The most closely related approach to ours is Melocoton [14], a program logic for reasoning about interactions between OCaml and C code via OCaml's FFI. Like our work, it addresses the challenge of relating high-level language values to low-level C representations and proving correctness of manually managed memory. However, the two approaches differ in methodology: Melocoton defines a new common program logic that integrates the separation logics of both languages, allowing assertions from both sides to coexist and enabling bidirectional transitions between views of the same value through *view reconciliation*. In contrast, our approach preserves existing program logics and verification tools as they are, introducing an explicit asymmetry in which reasoning proceeds from the more

abstract Python level to the C level. This means that Python-level reasoning uses only Nagini assertions, while C-level reasoning combines VeriFast specifications with predicates representing higher-level Python abstractions. While Melocoton formalizes its semantics and proves soundness foundationally in Rocq, our focus is on the practical integration of existing automated verification tools.

To the best of our knowledge, deductive verification of Python modules written in C has not been attempted before. The only tool targeting a similar setting is a static analysis based on abstract interpretation [18], which analyzes both Python and its connected C code to automatically detect common errors by building on independent analyses for each language.

Our observation that a significant challenge arises from the abstraction gap between Python and C is a recognized issue when reasoning about Foreign Function Interfaces (FFIs) [13]. We believe our strategy applies to similar settings, such as verifying native code called via the Java Native Interface (JNI) [17]. While JNI provides a more clearly defined interface than Python C modules and requires less direct manipulation of VM-internal data structures, it poses similar verification challenges; for example, native code must explicitly manage references to prevent the JVM from garbage-collecting active objects.

Finally, while our solution for reasoning about reference counting is tailored to this specific setting, general reasoning for reference counting in Separation Logic has been explored in various tools and logics, including VeriFast [15].

7 Conclusion and Future Work

We have presented the first framework for cross-language deductive verification between Python and C. While our current technique covers a sufficient subset of the Python C API to verify common "glue code", several avenues for future research remain. For instance, while we currently verify individual functions called from Python, C code can also define custom classes and perform callbacks into Python; supporting these behaviors is a key next step. Furthermore, while our C API specifications capture the proof obligations described in the documentation, future work could formally model CPython's internals to foundationally prove the correctness of these specifications.

Acknowledgments. We thank the ISoLA reviewers for their helpful comments. This work was inspired by discussions at the Lorentz Center seminar "Contract Languages" in 2024 and Schloss Dagstuhl seminar 26031 "Software Contracts Meet System Contracts" in 2026.

References

1. Armborst, L., Lathouwers, S., Huisman, M.: Joining forces! Reusing contracts for deductive verifiers through automatic translation. In: Herber, P., Wijs, A. (eds.) iFM 2023 - 18th International Conference, iFM 2023, Leiden, The Netherlands,

November 13-15, 2023, Proceedings. pp. 153–171. Lecture Notes in Computer Science, Springer (2023). https://doi.org/10.1007/978-3-031-47705-8_9
2. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. LNCS, vol. 4111, pp. 364–387. Springer (2005)
3. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods. pp. 127–131. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
4. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS. Lecture Notes in Computer Science, vol. 2694, pp. 55–72. Springer (2003)
5. Chen, P., Lin, S., Godbole, A., Singh, R., Polgreen, E., Lee, E.A., Seshia, S.A.: Polyver: A compositional approach for polyglot system modeling and verification. In: Irfan, A., Kaufmann, D. (eds.) Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design, FMCAD 2025, Menlo Park, CA, USA, October 6-10, 2025. TU Wien Academic Press (2025). https://doi.org/10.34727/2025/ISBN.978-3-85448-084-6_10
6. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskał, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics (TPHOLs). Lecture Notes in Computer Science, vol. 5674 (2009)
7. Eilers, M., Müller, P.: Nagini: A static verifier for Python. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. pp. 596–603. Lecture Notes in Computer Science, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_33
8. Ernst, G., Murray, T.: SecCSL: Security concurrent separation logic. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. pp. 208–230. Lecture Notes in Computer Science, Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_13
9. Ernst, G., Pfeifer, W., Ulbrich, M.: Contract-LIB: A proposal for a common interchange format for software system specification. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part III. pp. 79–105. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-75380-0_6
10. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: ESOP. LNCS, vol. 7792, pp. 125–128. Springer (2013)
11. Fonseca, P., Zhang, K., Wang, X., Krishnamurthy, A.: An empirical study on the correctness of formally verified distributed systems. In: Alonso, G., Bianchini, R., Vukolic, M. (eds.) Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017. pp. 328–343. ACM (2017). https://doi.org/10.1145/3064176.3064183
12. Foundation, P.S.: Python/C API reference manual, https://docs.python.org/3/c-api/index.html, accessed on April 18, 2026
13. Furia, C.A., Tiwari, A.: Challenges of multilingual program specification and analysis. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceed-

ings, Part III. pp. 124–143. Lecture Notes in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-75380-0_8
14. Guéneau, A., Hostert, J., Spies, S., Sammler, M., Birkedal, L., Dreyer, D.: Melocoton: A program logic for verified interoperability between OCaml and C. Proc. ACM Program. Lang. **7**(OOPSLA2), 716–744 (2023). https://doi.org/10.1145/3622823
15. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. pp. 41–55. Lecture Notes in Computer Science, Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
16. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015)
17. Liang, S.: The Java native interface: programmer's guide and specification. Addison-Wesley Professional (1999)
18. Monat, R., Ouadjaout, A., Miné, A.: A multilanguage static analysis of Python programs with native C extensions. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings. pp. 323–345. Lecture Notes in Computer Science, Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_16
19. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. pp. 41–62. Lecture Notes in Computer Science, Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
20. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
21. van Rossum, G., Lehtosalo, J., Langa, Ł.: Type Hints. https://www.python.org/dev/peps/pep-0484/ (2014)
22. Sammler, M., Spies, S., Song, Y., D'Osualdo, E., Krebbers, R., Garg, D., Dreyer, D.: DimSum: A decentralized approach to multi-language semantics and verification. Proc. ACM Program. Lang. **7**(POPL), 775–805 (2023). https://doi.org/10.1145/3571220
23. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012). https://doi.org/10.1145/2160910.2160911
24. Yang, H., Nong, Y., Wang, S., Cai, H.: Multi-language software development: Issues, challenges, and solutions. IEEE Trans. Software Eng. **50**(3), 512–533 (2024). https://doi.org/10.1109/TSE.2024.3358258