PyXray: Practical Cross-Language Call Graph Construction through Object Layout Analysis

Georgios Alexopoulos
University of Athens
Athens, Greece
grgalex@ba.uoa.gr

Thodoris Sotiropoulos
ETH Zurich
Zurich, Switzerland
theodoros.sotiropoulos@inf.ethz.ch

Georgios Gousios
Endor Labs
Palo Alto, California, USA
gousiosg@endor.ai

Zhendong Su
ETH Zurich
Zurich, Switzerland
zhendong.su@inf.ethz.ch

Dimitris Mitropoulos
University of Athens
Athens, Greece
dimitro@ba.uoa.gr

Abstract

A great number of software packages combine code in high-level languages, such as Python, with binary extensions compiled from low-level languages such as C, C++ or Rust to either boost efficiency or enable specific functionalities. In this context, high-level function calls can trigger native (binary) code execution. This setup introduces challenges for call graph generation. Accurate call graphs are essential for various applications, including vulnerability management and software maintenance, as they help track execution paths, assess security risks, and identify unused or redundant code.

This work tackles the problem of cross-language call graph construction in Python. Instead of relying on static analysis, which struggles with identifying Python-native interactions, we propose a dynamic analysis technique which does *not* require inputs to execute code. Our approach is based on two key insights: (1) when a binary extension is imported from Python code, all its objects (e.g., functions) are loaded into memory, and (2) the layout of callable Python objects contains pointers to the native functions they invoke. By analyzing these memory layouts for every loaded object, we identify corresponding graph edges, which link Python functions to the native functions they eventually invoke. This is an essential element for constructing call graphs across language boundaries.

We implement this approach in PyXray, a tool that efficiently analyzes massive Python packages such as NumPy and PyTorch in minutes, while significantly outpeforming existing static analysis methods in terms of precision and recall. PyXray enables two key applications: (1) cross-language vulnerability management, by identifying whether a Python package potentially calls a vulnerable native function and (2) cross-language bloat analysis, by quantifying unnecessary code across Python and native components.

1 Introduction

Software Composition Analysis (SCA) [24, 26, 27, 53] has become essential for managing security risks that originate from the extensive reuse of open-source software [19]. As software dependencies grow in number and complexity, understanding not just what components are included but also which ones are executed becomes paramount for effective vulnerability management [19, 21, 36, 43]. *Reachability analysis via call graphs* [20, 29, 42] provides this insight by identifying execution paths from application code to third-party functions. This allows developers to determine whether a third-party vulnerability poses a real threat to their application. Beyond security, reachability analysis is also valuable for software maintenance, as it helps detect *software bloat* [14, 25, 47]: if a dependency is imported (directly or transitively) and is not reachable, it can be removed, reducing storage requirements and runtime overhead.

Traditional SCA tools focus primarily on a single language (e.g. C [27, 53] or Java [26]). However, studies (including ours) indicate that 60–75% of the uncompressed size of popular package repositories is binary files [49]. This suggests that current reachability approaches that focus on a single language cover only a small portion of the code that may execute.

In this work, we focus on Python. Python packages frequently incorporate binary components implemented in languages such as C, C++, or Rust to optimize performance or access lower-level system functionalities [22, 34, 37]. This complicates call graph construction (and by extend reachability analysis), as traditional analysis tools operate on either the Python or binary level, but not across both [11, 12, 46, 54]. This gap stems from several factors, such as diverse languages used for implementing binary components, or variety of binding frameworks that connect Python with native code. Without cross-language reachability analysis, developers cannot determine which native components are actually used, leading to bloated applications that consume excessive storage and memory resources. At the same time, security teams waste resources addressing vulnerabilities in unused code, potentially overlooking critical issues in reachable components.

Approach: To address these challenges, we design a dynamic analysis that identifies the connections between Python callable objects and their native implementations. Our dynamic analysis does not require executing code with concrete inputs. Instead, it leverages key observations about Python's object system: when a binary extension is imported, all its callable objects are loaded into memory with their memory layouts containing pointers to the native functions they invoke. By systematically analyzing these memory layouts, our approach extracts the corresponding native function pointers and resolves them to binary symbol names. Combining Python call graphs, native code call graphs and the identified bridge points, our approach enables reachability analysis across Python and native code.

We evaluate our implementation, PyXray, by identifying the connections between Python and native code in widely-used Python

packages, such as `numpy` and `pytorch`. Additionally, we demonstrate how PyXray helps (1) prioritize vulnerability fixes and (2) detect unused binary dependencies in Python applications

Results: Our results indicate that PyXray achieves 100% recall in detecting Python-Native bridges across diverse packages, significantly outperforming existing static analysis approaches [22]. When applied to vulnerability analysis, PyXray reveals that while many applications depend on vulnerable packages, only a fraction (13%) invoke the vulnerable functions. Notably, our findings have led the developers of 10 clients affected by real-world vulnerabilities to upgrade their dependencies to safer versions. Our bloat analysis uncovers that binary components contribute substantially to installation size (63% on average), with roughly 92% of binary functions being unreachable from the applications that include them.

Contributions: Our work makes the following contributions:

- A novel and practical dynamic analysis technique that identifies Python-Native bridges. These bridges enable cross-language call graph construction and reachability analysis. (Section 3).
- An open-source implementation called PyXray (Section 4).
- A comprehensive evaluation on massive and popular Python packages, demonstrating PyXray's effectiveness (Section 5).
- Empirical insights into PyXray's impact on the Python ecosystem, enabling cross-language reachability analysis for vulnerability management incorporating 35 CVEs [1] and software bloat quantification for 984 client applications (Section 5).

2 Background and Motivation

Terminology: Python follows a design where everything is an *object*, including modules (imported source files), functions, or even types. An object may contain other object members, which can be accessed through their name (e.g., `obj.x`).

Python allows extending its functionality using *binary extensions*, which are compiled binary files distributed with a Python package. These binary files enable programmers to define their own built-in types and modules, which can be imported by the interpreter just like typical Python source files. When loaded, they become module objects containing functions, classes, and other members.

A *Python package* deployed in PyPI (Python Package Index) is a collection of Python source files, binary extensions, metadata, and dependencies. PyPI package distributions contain only the compiled binaries of their extensions, not the original source code (e.g., C code). From now on, the term "Python package" refers specifically to a distributed package in PyPI, rather than a collection of modules sharing the same namespace [44].

Running example: To motivate and explain the concepts behind our work, we introduce the code excerpts included in Figure 1, and use them as a reference throughout the paper. Consider a Python package called `thumbor`, which offers thumbnail services. The package consists solely of Python source files, one of which is `thumbor/utils.py` whose (oversimplified) code is shown in Figure 1a (lines 1–4). To implement its functionality, `thumbor` depends on another Python package named `Pillow`, which offers fast image processing features. As shown in Figure 1b (lines 1–5), the distribution of `Pillow` is a mix of of Python source files (`PIL/ImgCms.py`) and binary extensions (`PIL/_imagingcms.so`).

The `thumbor` package defines the method `ensure_srgb`, which processes an image by calling functions from `Pillow`, including the constructor of the `ImgCmsTransform` class (Figure 1a, lines 14–18). This constructor takes an image and, based on additional parameters, calls either `buildTransform` or `buildProofTransform` from the `_imagingcms` module (Figure 1b, lines 12–16). However, these functions are not implemented in Python; they are written in C. The `_imagingcms` module originates from the shared library `_imagingcms.so`, which is compiled from the corresponding C source file (Figure 1c). This C file defines two functions named `cms_build_transform` and `cms_build_p_trans` (lines 4–9), both of which internally call `cms_transform_new` (lines 5, 8). Using the CPython[1] API, the aforementioned C functions are ultimately exposed as function objects when importing the `_imagingcms` module from Python (lines 10–13 and 15–20). For example, the C function `cms_build_transform` is exposed as the function `buildTransform` in Python (Figure 1b: line 14, Figure 1c: line 11).

Call graph and its applications: Figure 2 shows the unified call graph for the above example. It consists of two parts: (1) the Python side (blue frame), which captures all caller-callee relationships within Python source files of `thumbor` and `Pillow`, and (2) the binary side (red frame), which captures function calls within the binaries (e.g., `_imagingcms.so`).

There are approaches [14, 28] that build *inter-package* Python call graphs, such as the one shown in the blue frame of Figure 2. These graphs track function calls across package boundaries, such as from `thumbor` to its dependency `Pillow`. However, these methods fail to bridge the gap between Python and native binaries. Their call graphs stop at functions like `_imagingcms.buildTransform` because they analyze only the Python side.

Detecting call edges that connect Python and binary call graphs is important for many applications, one of which is vulnerability management. For example, `cms_transform_new` in Figure 1c is affected by CVE-2024-2819. A call graph that unifies Python and native code can determine whether a dependent Python package is (transitively) affected by this vulnerability. In our example, there is a call path from `thumbor` (`ensure_srgb`) to the vulnerable function `cms_transform_new` (gray node in Figure 2). Thus, the developers of `thumbor` can use this information to prioritize upgrading to a safer Pillow version.

Problem statement: Given a Python call graph and a binary call graph produced by state-of-the-art call graph generators [38, 46], *the goal of this paper is to identify missing edges that connect them* (e.g., red edges in Figure 2). These edges enable the construction of a *unified call graph* (Figure 2). We refer to these edges as *Python-Native bridges* (or bridges for simplicity). A bridge is a triple $k \in Bridge = \langle n, s, b \rangle$, where n is a fully qualified Python identifier, such as `_imagingcms.buildTransform`, s is the name of the native function executed when calling n (e.g., `cms_build_transform`), and b is the binary file where s is defined (e.g., `PIL/_imagingcms.so`).

Challenges: One way to solve the bridge identification problem is to statically extract bridges from source code. For example, a static analyzer could examine the interactions of the C code in Figure 1c

[1] CPython is the standard implementation of Python.

```
thumbor/
|-- __init__.py
|-- utils.py
|-- ...
-----------------------------------------
# utils.py
from PIL import ImgCms

def get_profile(icc):
  # do something with img and return profile
  ...
  return profile

def ensure_srgb(img):
  prof = get_profile(img)
  transform = ImgCms.ImgCmsTransform(prof)
  ...
  return transform.apply(...)
```

(a) The thumbor PyPI package, which provides imaging services.

```
PIL/
|-- __init__.py
|-- ImgCms.py
|-- _imagingcms.so
|-- ...
-----------------------------------------
# ImgCms.py
from . import _imagingcms
core = _imagingcms

class ImgCmsTransform:
  def __init__(self, input, proof=None):
    if proof is None:
      core.buildTransform(...)
    else:
      core.buildProofTransform(...)

  def apply(self, args):
    ...
```

(b) The Pillow libray for image processing.

```
// _imagingcms.c
#include "Python.h"
static PyObject* cms_transform_new(...) {  }
static PyObject* cms_build_transform(...) {
 return cms_transform_new(...);
}
static PyObject* cms_build_p_trans(...) {
 return cms_transform_new(...);
}
static PyMethodDef pyCMSdll_methods[] = {
  {"buildTransform", cms_build_transform},
  {"buildProofTransform", cms_build_p_trans},
}

PyObject* PyInit__imagingcms(void) {
  static PyModuleDef module_def = {
    .m_name = "_imagingcms",
    .m_methods = pyCMSdll_methods,
  };
  return PyModule_Create(&module_def);
}
```

(c) The C extensions of the Pillow package.

Figure 1: A Python package (thumbor), which depends on another Python library, Pillow. Pillow's distribution includes a mix of Python and native code. Calling a function from thumbor triggers the execution of a C function in Pillow (CVE-2024-2819).

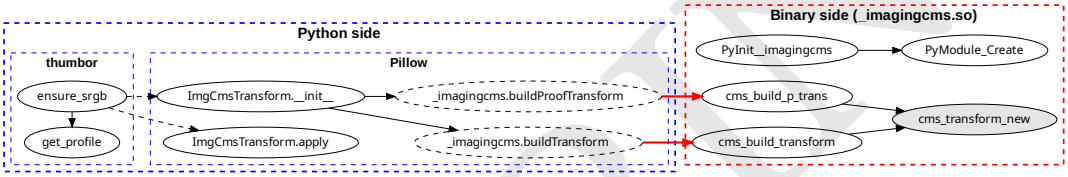

Figure 2: Unified call graph for the example of Figure 1.

with the CPython API and determine that `cms_build_transform` is the native implementation of the `buildTransform` Python object. Nevertheless, this involves several challenges:

- **Miscellaneous implementation languages**: Python packages use various languages for their binary extensions, not just C. For example, the Python package `cryptography` is written in Rust. The `numpy` package uses Cython (not CPython!) [7], which translates Python-like code into binary code. A static analysis needs to reason about all these different implementation languages.
- **Binding frameworks**: Unlike the example in Figure 1c, Python packages do not always interact directly with the CPython API when implementing extension modules. Instead, they rely on binding frameworks such as PyBind11 [50] and CFFI [45], which simplify native integration. This makes it challenging for static analysis to understand how these frameworks ultimately interact with the CPython API and extract the corresponding bridges.
- **Auto-generated source files**: In many cases, source files written in languages such as C/C++ are generated automatically at *build-time*, complicating static analysis.

3 Approach

We tackle the challenges of bridge identification (Section 2) with a *dynamic analysis* that does *not* require concrete inputs. Our approach leverages the insight that when a binary extension is imported (e.g., Figure 1b, line 8), *all* its objects (e.g., functions) are loaded into memory. By analyzing the memory layout of these objects, we extract the native functions invoked upon calling the objects. To crystallize these insights, we first provide technical background on CPython's object system and memory layout.

```
typedef PyObject *(*CFunction)(PyObject *, PyObject *);
typedef struct {
  // A pointer to the C function that implements the call
  CFunction ml_meth;
} MethodDef;
// Implementation of CFunctionObject
typedef struct {
  MethodDef *m_ml;
} CFunctionObject;
// Implementation of tp_call for CFunctionType
static PyObject *
cfunction_call(PyObject *func, PyObject *args, PyObject *kwargs) {
  CFunctionObject *obj = (CFunctionObject *) func;
  CFunction meth = obj->m_ml->ml_meth;
  PyObject *result = meth(args, kwargs);
  return result;
}
// Implementation of PyCFunctionType
typedef PyObject * (*ternaryfunc)(PyObject *, PyObject *, PyObject *);
typedef struct {
  size_t tp_size;
  ternaryfunc tp_call;
} TypeObject;

TypeObject CFunctionType = {
  sizeof(CFunctionObject), // tp_size
  cfunction_call // tp_call
};
```

Figure 3: Code snippet from CPython illustrating the implementation of the callable type `CFunctionType`.

3.1 Technical Background on Callable Objects

Callable objects and callable types: In Python, any object that can be called like a function using parentheses is considered *callable*. There are different types of callable objects, including regular functions defined in code via the `def` keyword, class methods and constructors, or built-in functions like `len()` or `print()`.

The key factor that determines whether a Python object is a callable is its *type*. In CPython, a type is represented internally as an instance of a C `struct` known as *type object*. These type

Table 1: List of CPython callable types along with the objects they produce. External packages (e.g., `numpy`) can implement their own callable types (shown below the dashed line). Callable objects are categorized as internal (I), wrapper (W), and foreign (F). The column "`Function pointer`" specifies the location within the callable object's layout, pointing to the native function that implements the callable's behavior.

Callable type	Derived callable object	Category	Function pointer
FunctionType	FunctionObject	I	-
MethodType	MethodObject	W	`extract(obj.im_func)`
InstanceMethodType	InstanceMethodObject	W	`extract(obj.func)`
StaticMethodType	StaticMethodObject	W	`extract(obj.sm_callable)`
GenericAliasType	GenericAliasType	W	`extract(obj.origin)`
ClassMethodDescrType	ClassMethodDescrObject	F	`obj.d_method.ml_method`
MethodDescrType	MethodDescrObject	F	`obj.d_method.ml_method`
ClassMethodDescrType	ClassMethodDescrObject	F	`obj.m_ml.ml_meth`
MethodWrapper	MethodWrapperObject	F	`obj.descr.d_wrapped`
WrapperDescrType	WrapperDescrObject	F	`obj.d_wrapped`
CFunctionType	CFunctionObject	F	`obj.m_ml.ml_meth`
UFuncType	UFuncObject (numpy)	F	`obj.functions`
CyFunctionType	CyFunctionObject (Cython)	F	`obj.m_ml.ml_meth`

objects contain various fields that define how their instances behave, including allocation, instantiation, and deallocation.

A key field in every type object is named `tp_call`, which is a function pointer that is executed when an instance is called like a function. In this context, when a Python object is called, CPython retrieves `tp_call` from its type and executes it. Callable objects originate from a type object whose `tp_call` is a valid function pointer, while non-callable objects (e.g., modules, integer values) have a type (e.g., `ModuleType`, `int`) where `tp_call` is `NULL`. For clarity, we refer to any type that produces callable objects as a *callable type*.

Categories of callable objects: We classify callables based on the behavior of the `tp_call` function associated with their types.

Internal callable objects: These correspond to callable objects for which the `tp_call` function invokes the Python interpreter to execute the bytecode associated with these objects. Callable objects of type `FunctionType`, such as a Python function defined via `def` or a lambda, belong to this category.

Foreign callable objects: These are callable objects whose layout contains a pointer to a native function which is in turn called by `tp_call` (to be elaborated in Figure 3). For example, all built-in functions (e.g., `print`) are objects whose invocation triggers the execution of a function implemented in native code (e.g., in C).

Wrapper callable objects: These are callable objects that are wrappers of other callable objects. The behavior of their `tp_call` is to invoke the `tp_call` of the wrapped object. For example, consider a `def` function named `m` defined inside a class `A`. The callable object `A().m` has type `MethodType`, which is a wrapper of the callable object `m` (with type `FunctionType`) and the receiver object (`A()`).

Callable types in CPython: CPython implements eleven callable types that create instances of callable objects. Table 1 categorizes them as internal (I), foreign (F), or wrapped (W) (ignore the "Function pointer" column for now). Figure 3 illustrates the implementation of one of them, that is, `CFunctionType`. This type object is used to create *foreign callable objects* that, when invoked from Python code, trigger the execution of native functions.

Such callable objects are implemented as instances of the `CFunctionObject` struct (Figure 3, lines 7–9). These instances are created by their type object (i.e., `CFunctionType`) whose `tp_call`

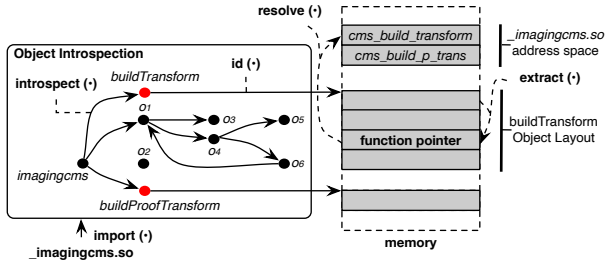

Figure 4: Object introspection and object layout analysis.

field points to function `cfunction_call` (lines 25–28). Just like every `tp_call` function, `cfunction_call` takes three arguments: the callable object that is called, along with its positional and keyword arguments (line 19). When an object of type `CFunctionType` is invoked, CPython executes `cfunction_call`. This function examines the structure of the callee `CFunctionObject` (first argument of `cfunction_call`), retrieves the corresponding function pointer that implements the call (line 14), and executes the underlying C function with the provided arguments (line 15).

An example of a foreign callable object of type `CFunctionType` is `_imagingcms.buildTransform` from Figure 1b. Internally, this function is represented as an instance of the `CFunctionObject` struct, which contains a pointer (Figure 3, line 4) to the native `cms_build_transform` function. When `_imagingcms.buildTransform` is called in Python, CPython invokes its `tp_call function`, which in this case is `cfunction_call` (Figure 3). This function inspects the `CFunctionObject`'s layout, retrieves the function pointer for `cms_build_transform`, and executes it.

Key insight: The discussion above leads us to a key observation: *the memory layout of every foreign callable object carries a pointer to a native function that is executed whenever the callable is invoked in Python.* This raises two questions: (1) How can we access this function pointer?, and (2) How can we do so systematically for every foreign callable object in a Python module?

To answer these questions, our approach leverages an *important insight*: when a binary extension is imported all its objects, including function objects, are loaded into memory. By systematically analyzing the layout of these objects, we extract the function pointers and determine the native functions they invoke.

Based on these observations and insights, we design a *dynamic analysis* that works as follows. Using Python's introspection capabilities, our approach exhaustively traverses an imported extension module, and identifies all callable objects included in it (*object introspection* step). For each identified callable object, the approach takes its *logical* memory address using the built-in Python function called `id()`. Since the memory region of foreign callable objects stores a pointer to the native function that is executed upon the object invocation (as per our previous observation), our approach examines the internal structure of the callable object at this address to locate this function pointer (*object layout analysis* step). Then, it resolves the symbol name of the underlying native function.

Approach with an example: Figure 4 illustrates how object introspection and object layout analysis work together to identify a bridge from Figure 1. Specifically, it shows how the Python object `buildTransform` is linked to the symbol `cms_build_transform`.

Algorithm 1: Finding C-Python bridges via object introspection

```
1  fun mod_introspect(b, n, threshold) =
2      obj ← import(b)
3      W ← [⟨n, obj⟩] ∪getLiveObjects()
4      bridges ← ∅
5      while W ≠ ∅ do
6          ⟨obj_name, obj⟩ ← pop W
7          segs ← split(obj_name, ".")
8          if len(segs) > threshold then continue
9          obj_type ← type(obj)
10         if isCallableType(obj_type) then
11             res ← layout_analysis(id(obj))
12             if res ≠ None then
13                 ⟨symbol, bin⟩ ← res
14                 bridges ← bridges ∪ {⟨obj_name, symbol, bin⟩}
15         for ⟨m_name, m_obj⟩ ∈ dir(obj) do
16             n ← obj_name + "." + m_name
17             W ← push ⟨n, m_obj⟩ to W
18     return bridges
```

- **Step 1. Import binary extension**: The binary extension `_imagingcms.so` is imported: all of its members (e.g., functions) are initialized and loaded into memory.
- **Step 2. Introspect module and identify callable objects**: recursively traverse the imported module using Python's introspection abilities and locate foreign callable objects within the module. There are two callable objects (shown with red bullets in Figure 4): `buildTransform` and `buildProofTransform`.
- **Step 3. Extract function pointer from the memory layout of the callable object**: take the memory address of the callable object given by the previous step (`buildTransform`) using Python's built-in function `id()`. Then, locate the specific memory slot within the layout of `buildTransform` where the function pointer of the callable is stored.
- **Step 4. Resolve symbol name**: resolve the symbol name of the function pointer extracted in the previous step. In our example, this leads to `cms_build_transform`.

In the following, we provide more details about the process of object introspection and object layout analysis.

3.2 Identifying Bridges

Identifying binary extensions: Our first step is to determine whether a Python package contains binary extensions. However, not all binaries in a Python package are extensions. A valid binary extension must implement an initialization function following the pattern `PyInit_<module-name>` like the one in Figure 1c (lines 15–20). To detect such extensions, the approach employs the `nm` Unix utility to analyze all binary files in a package and identify those containing a symbol that matches the aforementioned pattern. For example, the binary file `PIL/_imagingcms.so` initializes a module named `_imagingcms` (Figure 1c, line 15). For every identified binary extension file, our approach applies *object introspection*.

3.2.1 Object Introspection. Algorithm 1 provides an outline of our object introspection approach (let us ignore the shadowed lines for now). The algorithm (1) takes as input a binary extension b that initializes a Python module named n (line 1), and (2) returns a set of bridges. The algorithm begins by importing the given binary extension b. This initializes the corresponding module and its members, and loads them in memory. This import process results in the creation of a Python object *obj* that represents the imported module (line 2). To systematically explore the module's contents, the algorithm maintains a worklist W, which stores pairs of the form $\langle obj\_name, obj \rangle$. Each pair represents an object that has not yet been examined by the algorithm, along with the fully-qualified name of the object. The algorithm iterates through W and processes each object until all relevant objects in the module have been analyzed (lines 5–17). This exploration starts with the imported module itself, which serves as the root object (line 3).

At each iteration, the algorithm pops an element from W (line 6) and proceeds as follows. It first examines whether the type of the popped object is a callable type (lines 9–10) like the ones listed in Table 1. If so, the algorithm retrieves the memory address of the object using Python's `id()` built-in function and passes this address to the object layout analysis (more details in Section 3.2.2). The outcome of this layout analysis is the symbol name of the native function that is executed upon object invocation, along with the binary file where the symbol is located. When the result of object layout analysis is not `None` (meaning that the callable object is implemented via a native function), the algorithm adds the corresponding bridge to the list (lines 12–14).

Finally, the algorithm updates the worklist by adding all the children of the current object (lines 15–17). To do so, it leverages Python's built-in `dir()` function, which lists all member objects of a given object keyed by their names. Notably, before adding a new object to W (line 17), the algorithm constructs its fully-qualified name by concatenating the parent object's name with the child object's name (line 16). For example, the fully-qualified name of function object `buildTransform` is `_imagingcms.buildTransform`, as it is a member of a module object named `_imagingcms`.

Hidden objects: When importing a binary extension, certain objects might be "hidden". This refers to objects that are loaded into memory, but not directly exposed in the module. For example, in Figure 4, the object `o2` exists in memory but is not accessible from `_imagingcms`. To deal with hidden objects, we modify Algorithm 1 (line 3) to extend the initial worklist with all *live* objects in memory, including those not directly visible from the imported module.

Termination challenges: Algorithm 1 faces termination challenges that arise from two issues: circular references and object cloning. Some objects reference their ancestors, creating cycles that lead to infinite recursion. For example, in Figure 4, `_imagingcms.o6` references its ancestor `o1`, creating a cycle: `_imagingcms.o1.o4.o6.o1`. A possible solution is to track the memory addresses of the visited objects. However, this fails due to object cloning. When using `dir()`, some objects dynamically create *exact clones* of their ancestors at fresh memory addresses instead of referencing existing instances.

To prevent infinite loops, Algorithm 1 introduces a *threshold* parameter (line 1) that limits introspection depth. Before analyzing an object, the algorithm checks the length of its fully qualified name. If it exceeds the specified threshold, the object is ignored (line 8). This design decision is based on the observation that deeply nested callable objects (e.g., `a.b.c.d...`) are *rarely* invoked. Our

Algorithm 2: Extracting C function pointer from object.

```
1  fun extract(obj) =
2      obj ← cast(obj, PyObject)
3      obj_type ← obj.obj_type
4      match obj_type with
5          case FunctionType ⇒ return None
6          case MethodType ⇒
7              obj = cast(obj, PyMethodObject)
8              return extract(obj.im_func)
9          case CFunctionType ⇒
10             obj = cast(obj, PyCFunctionObject)
11             return obj.m_ml.m_meth
12         case … ⇒ …
```

evaluation shows that a threshold of 20 effectively analyzes all relevant objects while ensuring no bridges are missed (Section 5).

3.2.2 Object Layout Analysis. When Algorithm 1 finds a callable object, it passes its memory address as input to an object layout analysis (line 11). The result of this analysis is (1) the symbol name of the foreign function that is executed upon calling the given callable object, and (2) the binary file where the symbol is defined. To do so, the object layout analysis follows two steps.

Step 1: function pointer extraction: This step locates the function pointer within a callable object's layout, which varies based on its type. This is because different type objects create instances with distinct memory structures. For example, in a built-in callable object (e.g., `CFunctionObject`, produced by `CFunctionType`), the function pointer is stored in `obj.m_ml.ml_meth`, where obj is the callable's memory address (Figure 3).

We define `extract`, a function that takes a raw pointer to a callable object and retrieves a function pointer included in it. Algorithm 2 summarizes its logic. It determines the pointer's location based on the object type. If the object is a Python function (e.g., `FunctionType` created by keyword `def`), it does not have a function pointer, as CPython executes the function's bytecode instead (line 5). In this case, `extract` returns "None". If the object is a wrapper callable (e.g., `MethodType`), `extract` identifies the wrapped callable and recursively extracts its function pointer (lines 6–8). If the object is a foreign callable (e.g., `CFunctionObject`), `extract` retrieves the function pointer from its designated memory slot (e.g., `obj.m_ml.ml_meth`, lines 9–11). Note that Algorithm 2 covers only a subset of CPython's callable types. For a full reference, see the "Function pointer" column in Table 1.

Step 2: symbol name resolution: After extracting the function pointer from the callable object's layout, the next step is to resolve its symbol name and binary file. This is straightforwardly achieved by looking up the symbol table using GDB [18] (details in Section 4).

External callable types: While CPython defines a fixed set of callable types (Table 1), external Python packages can introduce custom callable types. To ensure our approach detects bridges, even if they stem from non-CPython callable objects, we work as follows.

At each iteration, Algorithm 1 checks whether a specific object is callable using *isCallableType* (line 10). This function essentially determines callability based on the layout of the object's type. In particular, it retrieves the memory address of the type object via `id()` and, using an object layout analysis similar to Algorithm 2, inspects its `tp_call` field. If `tp_call` points to a function outside the Python binary (e.g., `/usr/bin/python`), the type is considered *external*. This is because the `tp_call` implementation is not part of CPython and therefore unknown. Such external types produce callable objects beyond those listed in Table 1. When encountering an object of an unknown callable type, *isCallableType* issues a warning for further inspection.

While applying our technique to various Python packages (Section 5), we have identified three external callable types, including `UFuncType` from `numpy` and `CyFunctionType` from Cython (not CPython!). Table 1 lists their corresponding pointer extraction functions. Adding support for external types ensures the *completeness* of our object layout analysis: given an address of a callable object, the analysis is guaranteed to locate the native function implementation.

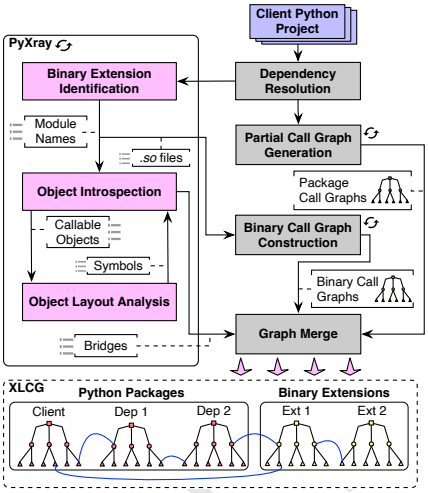

Figure 5: Building cross-language call graphs (XLCGs).

3.3 Building Cross-Language Call Graphs

Having introduced our method for identifying bridges (Section 3.2), we now describe how these bridges are used to construct *cross-language call graphs (XLCGs)*. Our approach for cross-language call graph construction is shown in Figure 5. Given a Python package, we resolve its dependencies, and perform the following steps:

Build Python call graphs: This step generates the Python call graph for the given Python package and its dependencies using a state-of-the-art Python call graph generator [46]. This results in multiple disjoint call graphs: one for each package and dependency. In the example of Figure 1, this step produces the two call graphs shown in the individual blue frames of Figure 2. Note these graphs are disconnected: they lack the dashed edges that connect them.

Build binary call graphs: This step locates binary artifacts within the given package and its dependencies. For each binary, it employs a state-of-the-art call graph generator [38] to construct the corresponding call graph. In the example of Figure 1, only the `Pillow` dependency includes a binary file (`PIL/_imagingcms.so`). The call graph for this binary is shown in Figure 2 (red frame).

Identify Python-Native bridges: This step considers only those packages in the dependency tree that include binary

extensions. It analyzes each of them as described in Section 3.2 to identify the corresponding bridges. In Figure 1, this step analyzes only `Pillow` since `thumbor` does not include binaries. The result is a set B that contains two bridges: (1) ⟨`buildTransform`, `cms_build_transform`, b⟩, and (2) ⟨`buildProofTransform`, `cms_build_p_trans`, b⟩, where b represents the binary file of `Pillow` (`PIL/_imagingcms.so`).

Merge Python and binary call graphs: This step produces the final XCLG of a Python dependency tree using two key inputs: (1) the individual Python and binary call graphs generated in previous steps and (2) the identified bridges. The process consists of two phases: first, it merges all individual Python call graphs into a unified Python call graph (blue frame in Figure 2); second, it integrates the unified Python call graph with the binary call graph using the identified bridges. To unify the individual Python call graphs, our approach uses state-of-the-art stitching algorithms [14, 28]. These algorithms integrate missing edges (e.g., black dashed edges in Figure 2). However, they do not consider the red edges that link Python and binary call graphs.

To add these missing red edges, our merging process examines the *leaf* nodes in the unified Python call graph and checks if they match any identified bridges. If a match exists, it connects each leaf node to its corresponding native function in the binary call graph (red edges). The result is a XLCG that captures caller-callee dependencies across Python and native binaries (Figure 2).

For example, in Figure 2, the Python call graph (the graph inside the blue frame) has four leaf nodes, but only two (marked with dashed borders) exist in the bridge set B (see step above). Our approach links each of these nodes to its corresponding binary function, and creates the red edges. For example, `buildTransform` in Python is linked to `cms_build_transform` in the binary.

4 Implementation and Discussion

We implement our approach for bridge identification (Section 3.2) as a command-line tool called PyXray, which consists of roughly 1k lines of Python code. The object introspection phase (Section 3.2.1) is implemented in pure Python using (1) its built-in functions (`id()`, `dir()`), and (2) the `gc` module to inspect all live Python objects in memory. *No* instrumentation is needed for this step. The object layout analysis is implemented via GDB [18], which PyXray attaches to itself and executes specific commands to extract key information as explained in Section 3.2.2. For example, the GDB commands used to extract the foreign function from any callable object of type `CFunctionType`, (e.g., `_imagingcms.buildTransform` in Figure 1b), are shown below. Here, 0x7f1a634840f0 represents the memory address of the callable object obtained using `id()`.

```
// cast address of callable
addr = (CFunctionObject *) 0x7f1a634840f0
fptr = addr->m_ml->m_meth // extract function pointer
symbol_info = info symbol fptr // resolve symbol name
```

Call graph generation: To build call graphs, we employ PyCG [46] and Ghidra [38]. However, note that our approach for constructing XLCGs (Figure 5) works regardless of the underlying call graph generators.

Limitations and assumptions: The current implementation of PyXray supports only ELF binaries on Linux distributions. Furthermore, while its dynamic approach ensures precision (no false bridges), it may miss bridges in certain cases, leading to false negatives. First, some callable objects might not be exposed in the module nor loaded into memory when importing the binary extension. This occurs when a callable object is dynamically created at runtime, e.g. within a higher-order function. We have encountered only two such cases in our evaluation (Section 5.3).

Second, a Python package can programmatically load any shared library (e.g., `libc`), not just binary extensions, using APIs from Python's standard library, such as `ctypes.CDLL`. PyXray is not designed to capture these calls.

Third, PyXray may miss callable objects of unknown types, i.e., types that are not part of CPython (Table 1). However, as discussed in Section 3.2.2, PyXray issues warnings when encountering unknown types. Additionally, our approach is extensible (see below), and enables new callable types to be incorporated when needed.

PyXray assumes that the binary extensions of the analyzed package are not stripped, meaning their symbols are available. Thankfully, most widely used PyPI distributions (e.g., `pytorch`, `numpy`) provide unstripped binaries. In rare cases where the binaries are stripped, we re-build them with debug symbols enabled.

Extensibility and Generalizability: PyXray can be easily extended to accommodate new callable types. This can be achieved by adding approximately 10–20 lines of code that extend Algorithm 2 and extract the function pointer from the layout of the external callable object. Our implementation already supports callable types that are defined in external packages, such as Cython, `numpy`.

PyXray supports only Python-Native bridges. However, it can be extended to other high-level languages, such as JavaScript. For example, Node.js' N-API [40] supports native addons, where a similar approach could inspect objects and their layout to identify native function calls. While this is future work, it represents a promising direction for expanding PyXray's capabilities.

5 Evaluation

We evaluate PyXray based on the following research questions:

- **RQ1** How effective is PyXray in identifying bridges? (Section 5.1)
- **RQ2** How does PyXray compare to existing approaches? (Section 5.2)
- **RQ3** Can PyXray help security teams with prioritizing vulnerability fixes? (Section 5.3)
- **RQ4** What percentage of binary-level code included in Python packages can be considered "bloat"? (Section 5.4)

5.1 RQ1: Effectiveness of PyXray

Setup: We evaluate PyXray's effectiveness by applying it to PyPI packages with at least one binary extension. Our benchmark consists of packages from two sources. First, we include four packages previously analyzed by other bridge-identification approaches [22]. Second, we select six additional packages from the list of *most* downloaded PyPI packages [23]. Our final benchmark list contains well-established Python packages, such as `pytorch`, `numpy`, and `cryptography` (Table 2).

For each package, we measure the number of native functions found in binary extensions and exposed as Python callable objects upon importing these extensions. We also track analysis time and

Table 2: Descriptive characteristics of the selected benchmarks and PyXray's execution results, including recall and execution time. PyXray is compared against a static analysis tool called Frog [22]. The "Framework" column indicates whether the package directly interacts with the CPython API ("raw") or uses a binding framework. Shadowed rows denote cases where "Ground truth" and "Recall" are based on a subset of source files rather than the full codebase.

					PyXray						Frog		
Package	Ext. binaries	Frameworks	Languages	Native funcs (Gr. truth)	Found	Recall (%)	Time (s)	Objects	Callable obj.	Foreign callable obj.	Found	Recall (%)	Precision (%)
pytorch	2	raw, Pybind11, CFFI	C, C++	353	7,871	100	107	16.0M	11.4M	477.1K	2,990	37.9	48.3
pyaudio	1	raw	C	48	48	100	1	325.7K	207.0K	20.3K	28	58.3	100
python-ldap	1	raw	C	29	29	100	4	1.1M	750.9K	68.9K	28	96.5	100
trace-cruncher	3	Cython	C, Cython	139	139	100	5	1.1M	724.1K	60.9K	82	58.9	100
numpy	19	raw, Cython, CFFI	C, Fortran C++, Cython	421	2,072	100	58	7.9M	4.9M	455.8K	-	-	-
PyYAML	1	Cython	Cython	28	28	100	2	802.8K	591.9K	28.6K	-	-	-
PyNaCl	1	CFFI	C	214	214	100	1	392.6K	266.3K	19.2K	-	-	-
cryptography	1	PyO3, CFFI	Rust	696	696	100	2	435.7K	298.8K	21.7K	-	-	-
pandas	44	Cython	C, Cython	146	2,380	100	1087	286.9M	215.6M	11.6M	-	-	-
grpcio	1	Cython	Cython	261	261	100	3	1.2M	866.0K	37.0K	-	-	-

the total objects examined during the introspection phase (Section 3.2.1). We do not compute precision, as PyXray's dynamic analysis ensures *no false positives*: every reported native function is directly extracted from an object's memory layout.

We perform this single-threaded experiment on a machine with a 3.6 GHz processor and 24GB of RAM.

Ground truth: To establish the ground truth, we *manually* inspect the codebase of each package and identify source files (e.g., C, Rust) that implement binary extensions, while ensuring that *no* additional files are dynamically generated at build time. Using this process, we exhaustively analyze seven out of ten packages. This is because the remaining packages (i.e., `pytorch`, `numpy`, `pandas`) are quite large (thousands lines of code), thus, full manual inspection is infeasible.

For these three large packages, instead of fully inspecting their codebase, we focus on source files that implement binary extensions without using binding frameworks. Binding frameworks (e.g., CFFI, Cython) provide a uniform API for defining Python-Native bindings, as they abstract away interactions with the CPython API. This means that if PyXray correctly captures bridges in one instance of a framework, it does so for any package that utilizes the framework. Therefore, for large packages, we only examine source files that directly interact with the CPython API, without these frameworks (e.g., as shown in Figure 1c). For `pandas`, which exclusively uses Cython, we randomly sample three of its source files.

The ground truth for each package is shown in the column "Native funcs (Ground truth)" of Table 2. Shadowed rows indicate that the ground truth is related to the aforementioned subset of source files. After adding support for external callable types from Cython and `numpy` (Section 4), PyXray no longer encounters unknown callable types when analyzing the benchmarks of Table 2.

Results: Table 2 presents the analysis results of PyXray. In all cases, PyXray achieves 100% recall, meaning it does not miss any Python-Native bridges included in the ground truth. Our results validate the core insight behind PyXray: when a binary extension is imported, all its function objects are loaded into memory. Therefore, PyXray can capture all of them, leading to no observed false negatives in practice, even if they are possible in theory.

Our results further indicate that PyXray is efficient. It analyzes most packages in less than 5 seconds and handles large packages, such as `numpy` and `pytorch` in under two minutes. The only exception is the analysis of `pandas`, which takes approximately 18 minutes. This slowdown occurs because `pandas` has many dependencies (e.g., `scipy`, `numpy`, `seaborn`, `matplotlib`), leading to a significantly higher number of objects being initialized and loaded into memory when importing its binary extensions. For example, during object introspection, PyXray examines 286.9 million objects in `pandas`, of which 215.6 million are callable objects.

5.2 RQ2: Comparison with Existing Approaches

Setup: We compare PyXray with Frog [22], a static-based approach for identifying Python-Native bridges. Frog statically analyzes C source files to find interactions with the CPython API or binding frameworks. Based on this analysis, it extracts bridges, which are then matched against callee functions identified from the analysis of Python code. The implementation of Frog is neither open source nor available to us. Therefore, we compare PyXray against Frog's results for the four packages analyzed in its original paper [22] (Table 2).

Results: Table 2 shows that PyXray outperforms Frog both in terms of recall and precision. For `pytorch`, Frog's precision and recall was reported by the authors only for a single source file. Regarding precision: false positives in Frog stem from the fact that it struggles to distinguish between Python callable objects that share a common function name, although they are different. For example, in `pytorch`, Frog fails to distinguish between the callables `torch.func` (where `torch` is a module) and `t.func` (where `t` is a tensor object). In contrast, PyXray's dynamic analysis ensures no false positives, as its object layout analysis is able to tell that the aforementioned objects are different (they point to different locations in memory).

Frog's low recall (avg: 63.6%) is due to its inability to handle diverse languages and binding frameworks used in Python packages (see "Challenges" in Section 2). For example, `pytorch` relies on both PyBind11 and CFFI, but Frog only supports PyBind11. Consequently, PyXray identifies 2.6× more bridges in `pytorch` (PyXray: 7,871 vs. Frog: 2,990). In contrast, PyXray's dynamic approach is agnostic to these implementation details. All callable objects ultimately share the same memory abstractions when loaded, regardless of the underlying binding frameworks and languages.

Table 3: Stats for the CVEs for which there is at least one client in our dataset that depends on the vulnerable package.

CVE	Package	Vuln. symbol	Bridges		Clients		
			Total	% vuln.	Depend	Call	Fixed
2020-10177	Pillow	`ImagingFliDecode`	71	1.4	29	0	N/A
2020-35654	Pillow	`_decodeStrip`	71	1.4	29	0	N/A
2020-5311	Pillow	`expandrow`	71	1.4	29	0	N/A
2021-34141	NumPy	`_convert_from_str`	2072	14.6	204	185	4/4
2021-25290	Pillow	`_tiffReadProc`	71	2.8	29	0	N/A
2022-30595	Pillow	`ImagingTgaRleDecode`	71	1.4	37	0	N/A
2023-25399	SciPy	`Py_FindObjects`	1203	0.1	122	4	3/4
2024-28219	Pillow	`cms_transform_new`	71	1.4	46	4	3/4

5.3 RQ3: Prioritizing Vulnerability Fixes

Setup: We demonstrate how PyXray can be leveraged for vulnerability management in a realistic setting. Specifically, we use cross-language reachability analysis in XLCGs to determine whether Python packages are affected by vulnerabilities in their native dependencies. This information helps security teams prioritize fixes by identifying which vulnerabilities are actually reachable from Python code. To do so, we rely on Common Vulnerability Enumerations (CVEs) [1]. We obtain CVEs along with the associated vulnerable (binary) functions from XXX's (redacted to preserve anonymity) proprietary, manually curated vulnerability database. We *share* the curated dataset as part of our replication package. In the following, we explain our setup.

Construction of XLCGs: We use the dataset from the work of Drosos et al. [14], which contains 1,302 Python applications from GitHub. From these, we consider 984 that have a corresponding PyPI release, and construct their corresponding XLCGs (Section 3.3).

Identification of CVEs: We identify CVEs that affect popular Python packages. Then we keep only entries that incorporate vulnerable functions (i.e. symbols) that reside in binary components of the package.

Reachability of vulnerable symbol: We assess the reachability of vulnerable symbols in both *internal* and *external* contexts. Internally, for the vulnerable package, we check if PyXray captures at least one bridge whose native function ultimately reaches the vulnerable symbol. Externally, we determine the extent to which clients from the set of 984 depend on vulnerable packages and simultaneously invoke vulnerable symbols, using the XLCGs of clients.

Overall, we obtain a dataset of (1) 35 CVEs and (2) 984 XLCGs for all applications (clients) in the original dataset.

Results: Out of the 35 CVEs, PyXray is able to discover bridges for 30 of them. For the remaining five cases: (1) the analyzed package is Windows-only (one case), (2) the foreign callable objects are created only after calling a specific function (two cases), and (3) the Python code programmatically loads the library, and calls its symbols, side-stepping Python's extension mechanism (two cases); currently PyXray is not designed to handle such cases (see "Limitations" in Section 4).

Out of the 30 CVEs, only 8 CVEs are related to vulnerable packages which in turn are dependencies for at least one of our 984 analyzed client applications. Table 3 summarizes the results for these 8 CVEs. Overall, PyXray unearths significant information for the developers of (1) vulnerable packages and (2) client applications relying on them, i.e., helps them efficiently estimate the impact of vulnerabilities and prioritize fixes as we discuss below.

Table 4: Bloat metrics for the 716 / 984 applications that have at least one dependency containing a binary extension.

Domain	Granularity	Statistical Measures				
		5%	Mean	Median	95%	Histogram
Python	Size	0.83 *MB*	**19.96 *MB***	8.83 *MB*	73.62 *MB*	
	Package (%)	0.00	**48.98**	50.00	86.67	
	File (%)	71.74	**87.54**	89.38	98.38	
	Function (%)	88.22	**95.52**	96.28	99.58	
Binary	Size	0.20 *MB*	**53.13 *MB***	12.66 *MB*	193.59 *MB*	
	File (%)	6.71	**67.41**	66.67	100.00	
	Function (%)	67.51	**92.26**	100.00	100.00	
Total	Package	0.00 *MB*	**37.75 *MB***	9.84 *MB*	160.95 *MB*	
	File	1.22 *MB*	**44.04 *MB***	17.06 *MB*	144.25 *MB*	
	Function	1.35 *MB*	**67.54 *MB***	20.70 *MB*	234.11 *MB*	

Package developers: As shown in the column "Bridges", PyXray reports the number of identified bridges ("Total") in the vulnerable package, and the percentage that leads to vulnerable code execution (those reaching the vulnerable symbol). When discovering a bug in a native function (before its fix or possible CVE assignment), package developers can use PyXray to estimate the number of transitively affected client applications. This way, they can prioritize bugs with higher reachability from Python callable objects. For example, in CVE-2021-25290 (`numpy`), the vulnerable function is reachable by 14.8% of the native functions exposed as objects in Python. In contrast, for CVE-2023-25399, only 0.1% of the `scipy` foreign callable objects lead to the vulnerable symbol.

Developers of client applications: The practical effect of this prioritization can be better seen in our analysis of client applications from our dataset. While many applications depend on vulnerable package versions ("Depends" column), only a few actually call the vulnerable native functions ("Calls" column). For example, although 122 clients depend on the vulnerable version of `scipy`, only four of them call the vulnerable function.

Across all analyzed CVEs, only three have at least one client where the vulnerable function is reachable. Our fine-grained reachability analysis reveals that a `Pillow` and a `scipy` CVE affect four clients each, while a `numpy` vulnerability impacts 185 clients. This discrepancy is due to the centrality of `numpy`'s vulnerable function within its API. Since 14.8% of its bridges lead to the vulnerable function, client applications are far more likely to invoke a `numpy` function that ultimately triggers the vulnerability.

Reporting issues: For each of the three CVEs, we submitted bug reports to four client applications transitively affected by the vulnerability. Each report included a recommended fix, suggesting an update to a safer dependency version. For example, in Figure 1, we advised `thumbor` to upgrade its Pillow dependency to a safer version. Overall, 10 / 12 issues have already been fixed as of now.

5.4 RQ4: Binary bloat in PyPI

The subject of *bloated dependency code* (or bloated code for short) [48] has attracted considerable research attention recently.

Bloated dependency code refers to the unused code that is incorporated into a software application through its dependencies. Past efforts [14] are confined to measuring bloat only at Python level. In this research question, we show how PyXray enables us to quantify bloated dependency code by taking into account both the Python and binary components of an application.

Setup: We reuse the 984 client applications and their XLCGs from RQ3, all of which are part of the recent study on bloated dependency code in PyPI [14]. First, we analyze the dependency tree of each client. Then, we identify binary files in the dependency tree and determine how many of their symbols are reachable from the client's Python functions. Unreachable symbols are marked as "bloated". If an entire binary file has no reachable symbols, it is also considered bloated. Based on these definitions, we quantify binary bloat using two granularities: binary files, and binary functions.

Table 4 presents the findings of our bloat analysis. Out of the 984 client applications, 716 of them have at least one dependency containing a binary extension. We now use the term "Python domain" to refer to bloat in pure Python code (`.py`), and "Binary domain" to refer to bloat in binary code (`.so`).

Binary bloat (file granularity): Overall, two thirds (67%) of binary files from an application's dependencies are entirely unused. Compared to the Python domain (file granularity), fewer binary files are bloated. This could be attributed to the fact that binary files generally contain more functions per file since they are compiled and linked from multiple source files.

To reduce installation size, we could take different measures based on the nature of the bloated binary file. If the bloated binary is an extension module, it could be replaced with a stub that only implements its initialization function (`PyInit_<module-name>`, Section 3.2). Otherwise, it could be removed entirely. This approach can reduce installation size by ~36MB, on average.

Binary bloat (function granularity): PyXray 's features enables us to measure bloat at function granularity. On average, only 7.7% of functions in dependency binaries are used. To reduce installation size, one can apply binary debloating techniques [4], thus keeping only the reachable functions. Assuming each function occupies a fixed amount of space in the code / text segment of the binary, such a measure could reduce installation size by approximately 49MB.

Notably, for clients of exceedingly large packages, such as `pytorch` (1.4GB binary size), the average bloated size is 943.7MB at the file level and 1.29GB at function level. In these cases, the need for active debloating is even more apparent.

Combining Python and binary bloat: PyXray allows us to compute the overall bloat by taking into account both the Python and the binary domains. Our results show that, on average, binary files make up 63% of a package's installed artifacts, contributing 2.3× more to the final installation size than Python files. The average Python package, including its dependencies, occupies 73.1MB, with a significant portion classified as bloat. At the package level, 37.8MB is bloated; at the file level, 44.0MB; and at the function level, 67.5MB.

6 Related Work

Cross-language analyses: The closest work to PyXray is Frog [22], a static analysis approach for identifying Python-Native bridges used in cross-language call graph construction. To extract bridges, Frog examines C source files. Nevertheless, as shown in Section 5.2, it suffers from false positives and a high number of false negatives due to the challenges discussed in Section 2.

Most research on Python-C interactions focuses on static analysis frameworks for detecting errors, such as arithmetic overflows and type errors across Python-C boundaries. These frameworks rely on either abstract interpretation [13, 37] or declarative analysis (e.g., CodeQL) [51]. Other work targets reference counting errors in Python objects due to incorrect native implementations [33, 35]. On the dynamic front, PolyCruise applies dynamic information flow analysis to detect vulnerabilities (e.g., buffer overflows) in Python packages with native extensions. Unlike our work, it requires concrete execution inputs.

Beyond Python-C, prior work explores static, cross-language analysis in Android [6, 10, 30], Node.js [9], and Java Native Interface (JNI) [17, 31, 41]. In contrast, PyXray takes a dynamic approach without relying on source code availability. Future work includes extending PyXray to other interpreted languages (e.g., JavaScript) that interact with binary artifacts.

Call graph construction: There are numerous tools for constructing call graphs in statically-typed languages, including doop [8], wala [16], cgc for Java, and Coral [12], Kelp [11] for C. For binaries, the reverse engineering frameworks Ghidra [38], Radare2 [3], and IDA Pro [2] provide tools for call graph generation and analysis.

Focusing on dynamic languages, prior work includes Python call graph generators [46, 54] that deal with complicated features, such as multiple inheritance and higher-order functions. In JavaScript, Feldthaus et al. [15] propose a flow-based analysis for unsound but accurate call graphs, while Nielsen et al. [39] present a modular approach for Node.js, incorporating package management internals for vulnerability analysis. Lehmann et al. [32] address WebAssembly's challenges (e.g., unmanaged linear memory).

All this work is orthogonal to our contributions: our approach for building cross-language call graphs (Section 3.3) allows the seamless integration of different Python and binary call graph generators.

Software ecosystem analysis: Hejderup et al. [21] propose using call graphs to map software ecosystems' calling relationships. Expanding on this, Mir et al. [36] analyze the Maven ecosystem, finding that while one-third of packages have vulnerable transitive dependencies, only 1% invoke a vulnerable method. Drosos et al. [14] examine bloated dependency code in PyPI. However, their analysis considers bloat only in Python files. Recent research has focused on security issues within scripting language ecosystems, particularly software supply-chain vulnerabilities. Alfadel et al. [5] show that vulnerabilities in PyPI packages often remain undetected for years. In the JavaScript ecosystem, Zahan et al. [52] analyze metadata from 1.6 million packages, identifying metrics that signal supply chain risks.

These software ecosystem analyses can benefit from PyXray by integrating both high-level (Python) and low-level (binary) code. Sections 5.3 and 5.4 demonstrate how PyXray enables cross-language reachability analyses in PyPI for vulnerability management and software bloat.

7 Conclusion

We presented PyXray, a practical tool for discovering the native function implementations of Python callable objects in PyPI packages with binary extensions. PyXray leverages a novel dynamic analysis, based on the key insight that when a binary extension is imported, all its callable objects (functions) are loaded into memory, embedding pointers to the native functions they invoke. By systematically analyzing these layouts, PyXray extracts the corresponding function pointers, and resolves them to the callee native functions. Linking Python callable objects with their native implementation enables cross-language call graphs for Python packages.

Our experiments demonstrate that PyXray is both efficient and effective, as it analyzes large packages, such as `numpy` and `pytorch` in minutes. Compared to existing static analysis approaches, PyXray achieves higher precision and recall. Furthermore, PyXray enables advanced cross-language reachability analyses in the PyPI ecosystem, particularly in vulnerability management and dependency bloat analysis.

References

[1] . 2025. CVE security vulnerability database. https://www.cvedetails.com/. Online accessed; 13-03-2025.

[2] . 2025. IDA Pro. https://hex-rays.com/ida-pro. Online accessed; 12-03-2025.

[3] . 2025. Radare2: Free Reversing Toolkit. https://rada.re/n/. Online accessed; 12-03-2025.

[4] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-scale Debloating of Binary Shared Libraries. *Digital Threats* 1, 4, Article 19 (Dec. 2020), 28 pages. doi:10.1145/3414997

[5] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in Python packages. *Empirical Software Engineering* 28, 3 (25 Mar 2023), 59. doi:10.1007/s10664-022-10278-4

[6] Sora Bae, Sungho Lee, and Sukyoung Ryu. 2019. Towards understanding and reasoning about Android interoperations. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 223–233. doi:10.1109/ICSE.2019.00038

[7] Stefan Behnel, Robert Bradshaw, David Woods, Matúš Valo, and Lisandro Dalcín. 2025. Cython: C-Extensions for Python. https://cython.org/. Online accessed; 12-03-2025.

[8] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. doi:10.1145/1640089.1640108

[9] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*. 559–578. doi:10.1109/SP.2017.68

[10] Achim D. Brucker and Michael Herzberg. 2016. On the Static Analysis of Hybrid Mobile Apps. In *Engineering Secure Software and Systems*, Juan Caballero, Eric Bodden, and Elias Athanasopoulos (Eds.). Springer International Publishing, Cham, 72–88.

[11] Yuandao Cai, Yibo Jin, and Charles Zhang. 2024. Unleashing the power of type-based call graph construction by using regional pointer information. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 78, 18 pages.

[12] Yuandao Cai and Charles Zhang. 2023. A Cocktail Approach to Practical Call Graph Construction. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 257 (Oct. 2023), 33 pages. doi:10.1145/3622833

[13] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

[14] Georgios-Petros Drosos, Thodoris Sotiropoulos, Diomidis Spinellis, and Dimitris Mitropoulos. 2024. Bloat beneath Python's Scales: A Fine-Grained Inter-Project Dependency Analysis. *Proc. ACM Softw. Eng.* 1, FSE, Article 114 (July 2024), 24 pages. doi:10.1145/3660821

[15] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 752–761.

[16] Stephen Fink and Julian Dolby. 2012. WALA—The T.J. Watson Libraries for Analysis.

[17] George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. 2020. Identifying Java calls in native code via binary scanning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 388–400. doi:10.1145/3395363.3397368

[18] Free Software Foundation, Inc. 2025. GDB: The GNU Debugger. https://www.sourceware.org/gdb/. Online accessed; 12-03-2025.

[19] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2019. A double-edged sword? Software reuse and potential security vulnerabilities. *Lecture Notes in Computer Science* (2019), 187–203. doi:10.1007/978-3-030-22888-0_13

[20] Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios. 2022. Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering* 27, 5 (2022), 102. doi:10.1007/s10664-021-10071-9

[21] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software Ecosystem Call Graph for Dependency Management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results* (Gothenburg, Sweden) *(ICSE-NIER '18)*. Association for Computing Machinery, New York, NY, USA, 101–104. doi:10.1145/3183399.3183417

[22] Mingzhe Hu, Qi Zhao, Yu Zhang, and Yan Xiong. 2023. Cross-Language Call Graph Construction Supporting Different Host Languages. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 155–166. doi:10.1109/SANER56733.2023.00024

[23] Hugo van Kemenade. 2025. Top PyPI Packages. https://hugovk.github.io/top-pypi-packages/. Online accessed; 10-03-2025.

[24] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. 2021. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '21)*. Association for Computing Machinery, New York, NY, USA, Article 5, 11 pages. doi:10.1145/3475716.3475769

[25] Rasoul Jahanshahi, Babak Amin Azad, Nick Nikiforakis, and Manuel Egele. 2023. Minimalist: Semi-automated Debloating of PHP Web Applications through Static Analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5557–5573. https://www.usenix.org/conference/usenixsecurity23/presentation/jahanshahi

[26] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 224, 13 pages. doi:10.1145/3597503.3639100

[27] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2023. Third-Party Library Dependency for Large-Scale SCA in the C/C++ Ecosystem: How Far Are We?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1383–1395. doi:10.1145/3597926.3598143

[28] Mehdi Keshani, Georgios Gousios, and Sebastian Proksch. 2023. Frankenstein: fast and lightweight call graph generation for software builds. *Empirical Software Engineering* 29, 1 (16 Nov 2023), 1. doi:10.1007/s10664-023-10388-7

[29] Seongmin Lee and Marcel Böhme. 2023. Statistical Reachability Analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 326–337. doi:10.1145/3611643.3616268

[30] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: static analysis framework for Android hybrid applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 250–261. doi:10.1145/2970276.2970368

[31] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 127–137.

[32] Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. 2023. That's a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 892–903. doi:10.1145/3597926.3598104

[33] Siliang Li and Gang Tan. 2014. Finding Reference-Counting Errors in Python/C Programs with Affine Analysis. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–104.

[34] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2513–2530.

https://www.usenix.org/conference/usenixsecurity22/presentation/li-wen
[35] Xutong Ma, Jiwei Yan, Hao Zhang, Jun Yan, and Jian Zhang. 2023. Detecting Memory Errors in Python Native Code by Tracking Object Lifecycle with Reference Count. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1429–1440. doi:10.1109/ASE56229.2023.00198
[36] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. 2023. On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 201–211. doi:10.1109/SANER56733.2023.00028
[37] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis*, Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi (Eds.). Springer International Publishing, Cham, 323–345.
[38] National Security Agency. 2025. Ghidra. https://ghidra-sre.org/. Online accessed; 10-03-2025.
[39] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node.js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 29–41. doi:10.1145/3460319.3464836
[40] Node.js Documentation. 2025. C/C++ Addons - N-API. https://nodejs.org/download/release/v8.9.4/docs/api/n-api.html. Online accessed; 10-03-2025.
[41] Jihee Park, Sungho Lee, Jaemin Hong, and Sukyoung Ryu. 2023. Static Analysis of JNI Programs via Binary Decompilation. *IEEE Transactions on Software Engineering* 49, 5 (2023), 3089–3105. doi:10.1109/TSE.2023.3241639
[42] Isaac Polinsky, Pubali Datta, Adam Bates, and William Enck. 2024. GRASP: Hardening Serverless Applications through Graph Reachability Analysis of Security Policies. In *Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 1644–1655. doi:10.1145/3589334.3645436
[43] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E. Santosa, Asankhaya Sharma, and David Lo. 2021. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Softw. Engg.* 26, 4 (July 2021), 34 pages. doi:10.1007/s10664-021-09959-3
[44] Python Software Foundation. 2024. Modules. https://docs.python.org/3.10/tutorial/modules.html. Online accessed; 17-02-2025.
[45] Armin Rigo and Maciej Fijalkowski. 2025. CFFI Documentation. https://cffi.readthedocs.io/en/stable/. Online accessed; 12-03-2025.
[46] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1646–1657. doi:10.1109/ICSE43902.2021.00146
[47] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A Longitudinal Analysis of Bloated Java Dependencies *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1021–1031. https://doi.org/10.1145/3468264.3468589
[48] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Softw. Engg.* 26, 3 (May 2021), 44 pages. doi:10.1007/s10664-020-09914-8
[49] Tom Forbes. 2025. The contents of PyPI, in numbers. https://py-code.org/stats. Online accessed; 12-03-2025.
[50] Wenzel Jakob. 2025. PyBind11 Documentation. https://pybind11.readthedocs.io/en/stable/index.html. Online accessed; 12-03-2025.
[51] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. 2023. Declarative static analysis for multilingual programs using CodeQL. *Software: Practice and Experience* 53, 7 (2023), 1472–1495. doi:10.1002/spe.3199 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3199
[52] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the npm supply chain?. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 331–340. doi:10.1145/3510457.3513044
[53] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. 2023. Software Composition Analysis for Vulnerability Detection: An Empirical Study on Java Projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 960–972. doi:10.1145/3611643.3616299
[54] Jiacheng Zhong. 2022. A practical call graph construction method for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1805–1807. doi:10.1145/3540250.3559077