The First Large-Scale Systematic Study of Python Class Pollution Vulnerability

Zhengyu Liu, Jiacheng Zhong, Jianjia Yu, Muxi Lyu, Zifeng Kang, and Yinzhi Cao
{zliu192, jzhong29, jyu122, mlyu4, zkang7, yinzhi.cao}@jhu.edu
Johns Hopkins University

Abstract—**Class pollution is a recently discovered, yet under-explored Python vulnerability that allows attackers to pollute unintended runtime objects by exploiting the class-based inheritance model and reflection mechanism. Before this paper, only two real-world vulnerabilities related to class pollution—including one reported to the Common Vulnerabilities and Exposures (CVE) database—were discovered. Furthermore, there was no existing tool capable of detecting such vulnerabilities, let alone a systematic study of vulnerable code patterns, exploitation techniques, and real-world prevalence.**

In this paper, we design and implement Pyrl, the first framework for detecting class pollution vulnerabilities in real-world applications via a novel, static *operational taint analysis*. Our key insight is that class pollution consists of two types of vulnerable code primitives—"get" and "set"—for fetching and setting items and attributes. Different combinations of these primitives (two types of "get"s and three "set"s) further lead to six unique vulnerability types according to our first taxonomy of class pollution. Pyrl's operational taint analysis tracks attacker-controlled inputs on these primitives and their combinations using fine-grained, operational taint labels that are initiated, transformed, propagated, and merged according to the analysis context.

We applied Pyrl to over half a million real-world Python programs from GitHub and PyPI, resulting in the detection of 47 zero-day, exploitable class pollutions. Our findings include critical vulnerabilities in widely used applications, such as Azure CLI by Microsoft and Mesop by Google, both of which have been acknowledged and patched. We have responsibly reported all identified vulnerabilities to the corresponding developers—who fixed five of them—and CVE Numbering Authorities (CNAs)—who assigned seven CVE identifiers.

1. Introduction

Python has become one of the most widely adopted programming languages with applications that span machine learning, data science, and software engineering. This widespread adoption is driven by the rich ecosystem of Python, with more than 630,000 packages available in the official package repository, PyPI [1], and more than one billion daily package downloads. Furthermore, the rise of Python-based Machine Learning (ML) libraries, such as PyTorch, also contributes to Python's popularity. For example, a recent survey showed that Python overtook JavaScript as the most-used language on GitHub in 2024 [2].

Python's easy-to-use characteristic comes from two fundamental design features: the uniform data model and the flexible reflection mechanism. Under Python's uniform data model, every value is treated as an object, whether it is a number, a string, a class, a function, or a module. Each object is associated with a set of built-in attributes, known as "dunder" (double-underscore) attributes, which specify its type hierarchy and metadata. The reflection mechanism, on the other hand, supports inspecting and modifying objects at runtime using dynamically determined names and values. This allows programs to easily adjust their structure and behavior during execution.

Unfortunately, the combination of these two fundamental features leads to a new vulnerability type, known as *class pollution* [3]. That is, an adversary leverages a sequence of reflective attribute lookups with attacker-controlled "dunder" names to traverse objects and modify attributes in unintended classes or modules. It was first introduced in 2023 in a blog post [3], which also disclosed a real-world vulnerability in the pydash library [4]. Since the initial release, there has been only one more vulnerability discovered and documented with a CVE [5] based on our search.

One major reason for such a limited number of known real-world vulnerabilities is the lack of any detection tools for class pollution, to the best of our knowledge. On the one side, existing Python vulnerability analysis tools [6]–[8] are designed to detect classic injection vulnerabilities with well-defined sinks (e.g., `eval`) as opposed to specific object assignments as sinks in class pollution. On the other side, many detection tools [9]–[15] are present for vulnerabilities that are similar to class pollution, particularly JavaScript prototype pollution, where properties are inherited and polluted via the prototype chain. However, those tools do not apply to Python class pollution due to fundamental differences in the programming languages, e.g., different inheritance models (prototype-based vs. class-based), object structures, attribute resolutions, and reflection mechanisms.

Beyond vulnerability detection, the study of Python class pollution in general is also limited in academic research, with only two recent works focusing on the aspects of exploitation and defense. In 2023, Ouyang [16] demonstrated the feasibility of class pollution attacks through a small, synthetic example. In 2024, Zhang [17] explored an exploitation technique targeting global variables pollution and discussed two possible defenses. However, both studies are less practical, relying on manually crafted examples and lacking analysis of real-world vulnerabilities. That said, thus far, prior works on class pollution have only scratched the surface, leaving out many important aspects, such as root causes, vulnerability patterns, real-world prevalence, and potential consequences.

In this paper, we design and implement the first static framework, named Pyrl (/*'Pearl'*/), to detect Python class pollution vulnerability via a novel technique, called operational taint analysis. Our key insight is that class pollution involves two types of vulnerable code patterns—defined as "get" and "set" primitives—for fetching and setting object attributes or collection items. The combination of these primitives—according to our novel taxonomy of class pollution—further leads to six different types of vulnerabilities with two types of "get" and three "set". Note that the literature on class pollution only discussed one out of the six, i.e., when both "get" and "set" primitives are unconstrained with unlimited read or write access of either object attributes or collection items, thus leaving the rest five as our novel contributions.

Our insight on "get" and "set" primitives also brings challenges in detecting class pollution vulnerabilities. The detection of "get" primitives requires the tracking of different fetched objects via either object attributes or collection items or both. That is, the same vulnerable code containing a "get" primitive may be used multiple times with the same or different branches for fetching attributes or items to reach the final target for class pollution. The detection of "set" primitives is similarly challenging, because the "set" operation could be an attribute assignment, an item assignment, or—in the most powerful case—a pair of both that operates on the same resolved object.

Pyrl's operational taint analysis tackles the challenge with a set of fine-grained, expressive labels, called operational taints. Specifically, operational taints are propagated on the target program with different operations (e.g., transforming, deriving, and merging) to track "get" and "set" primitives, leading to the detection of class pollution. For example, when a key in an attribute access is tainted, called a key taint, the resulting object obtains a so-called object taint denoted with the access type. The "get" primitives are tracked with the help of a novel label operator that merges object labels from disjoint control-flow paths with different access types. Then, the "set" primitives are identified using a pair of assignment tuples whose target objects share the same object label. Since object labels propagate under strict policy and capture the resolution semantics, their equivalence indicates that both assignments operate on the same attacker-resolved object.

We evaluated Pyrl on a large-scale dataset from two sources—GitHub repositories with over 100 stars [18] and all available packages from PyPI [1]—covering 671,475 projects. Our analysis revealed an alarming prevalence of class pollution vulnerabilities. In total, Pyrl reported 868 alerts, of which we manually verified 84 cases and confirmed 47 true positives. Among these, 15 can be triggered via local input, and 11 can be triggered remotely. We further analyzed the exploitability of these vulnerabilities and successfully demonstrated severe consequences, including remote code execution (RCE), credential leakage, cross-site scripting (XSS), and denial-of-service (DoS) in widely used applications. These include those maintained by Google and Microsoft, such as Mesop and Azure CLI. Both have since been acknowledged and patched. We responsibly disclosed all zero-day vulnerabilities to the corresponding vendors and have so far obtained seven CVEs.

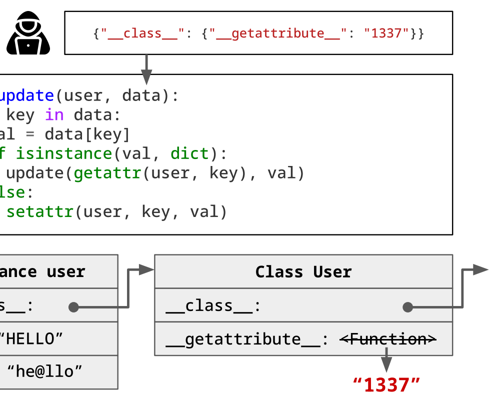

```
{"__class__": {"__getattribute__": "1337"}}
```
```
def update(user, data):
 for key in data:
   val = data[key]
   if isinstance(val, dict):
     update(getattr(user, key), val)
   else:
     setattr(user, key, val)
```

Figure 1: An example of class pollution vulnerability.

Contributions. In summary, this paper makes the following contributions.

- We design and implement Pyrl, the *first* automated tool for detecting class pollution vulnerabilities via a novel static technique called operational taint analysis, where taint is propagated with multiple operational semantic labels to precisely model class pollution behaviors.
- We establish the first systematic taxonomy of Python class pollution, revealing five previously unknown types of class pollution, one novel attack target, and the leading consequences, fundamentally expanding the threat landscape of this new vulnerability class.
- Pyrl detected 47 zero-day class pollution vulnerabilities across PyPI [1] and GitHub [18], with seven CVEs assigned so far. The results include those in popular projects such as Microsoft Azure CLI, Google Mesop, HuggingFace diffusers and accelerate, ComfyUI, and Taipy. Our findings show that class pollution can cause critical consequences, including token leakage, authentication bypass, and cross-site scripting, none of which had been previously demonstrated in real-world cases.

2. Class Pollution Vulnerability

In this section, we begin with the definition of class pollution vulnerabilities in §2.1, then present the first taxonomy of them in §2.2, and describe the threat model in §2.3.

2.1. Vulnerability Definition

A class pollution vulnerability occurs when attacker-controlled input leads to unintended modifications of dy-

TABLE 1: Systemization of reflected get operations in Python from builtins and standard libraries. "Features" describe the operation type, applicable data types, origin, and lookup order. "Capabilities" describe whether the lookup can access dunder names, methods, or other values. In the "Prevalence" column, "# Inst." and "# Pack." show the number of instances and packages where the get operation appears in the 50K most-downloaded PyPI packages. "Spec." records references to official documentation.

		Features				Capabilities			Prevalence		
	Code	Type	Apply	Origin	Order	Dunder	Method	Other	# Inst.	# Pack.	Spec.
#1	`getattr(obj, name)`	Attr	O/M/S	Builtins	First	●	●	●	719.1K	21.7K	[19]
#2	`obj.__dict__[name]`	Attr	O/M/S	Builtins	Second	◐	◐	●	15.3K	3.3K	[20]
#3	`obj.__getattribute__(name)`	Attr	O/M/S	Builtins	Second	●	●	●	13.1K	1.7K	[21]
#4	`vars(obj)[name]`	Attr	O/M/S	Builtins	Second	◐	◐	●	9.3K	1K	[22]
#5	`inspect.getmembers(obj)`	Attr	O/M/S	Inspect	Second	●	●	●	4.9K	1.9K	[23]
#6	`object.__getattribute__(obj, name)`	Attr	O/M/S	Builtins	First	●	●	●	4.3K	843	[21]
#7	`operator.attrgetter(name)(obj)`	Attr	O/M/S	Operator	Second	●	●	●	543	194	[24]
#8	`inspect.getattr_static(obj, name)`	Attr	O/M/S	Inspect	First	●	●	●	398	121	[25]
#9	`dir(obj)[index]`	Attr	O/M/S	Builtins	Second	●	●	●	93	23	[26]
#10	`inspect.getmembers_static(obj)`	Attr	O/M/S	Inspect	Second	●	●	●	11	8	[27]
#11	`dict[key]`	Item	M/S	Builtins	First	○	○	●	50.7M	44.1K	[28] [29]
#12	`dict.get(key)`	Item	M	Builtins	Second	○	○	●	8.6M	33.1K	[30]
#13	`dict.pop(key)`	Item	M/S	Builtins	Second	○	○	●	1.7M	17.9K	[30] [31]
#14	`dict.setdefault(key)`‡	Item	M	Builtins	Second	○	○	●	111.1K	8K	[32]
#15	`dict.__getitem__(key)`	Item	M/S	Builtins	Second	○	○	●	63.3K	3K	[33]
#16	`operator.getitem(dict, key)`	Item	M/S	Operator	First	○	○	●	271	79	[34]
#17	`operator.itemgetter(key)(dict)`	Item	M/S	Operator	Second	○	○	●	153	60	[35]
#18	`operator.__getitem__(dict, key)`	Item	M/S	Operator	First	○	○	●	3	1	[36]
#19	`eval(f"EXPR†", {"o": obj})`	Attr/Item	O/M/S	Builtins	First	●	●	●	331	135	[37]
#20	`exec(f"EXPR†", {"o": obj})`	Attr/Item	O/M/S	Builtins	First	●	●	●	172	112	[38]

Legend: O: Object; M: Mapping; S: Sequence; ●: Fully Supported; ◐: Conditionally Supported (#2 and #4 apply for class objects with `__dict__`); ○: Not Supported; ‡: `dict.setdefault` returns the value if key exists. EXPR†: Any operation from #1–#18 where only the key name is attacker-controlled and all other expressions are sanitized.

namically determined objects via Python's class-based inheritance model. We use the `update` function in Figure 1 as an illustration. This function is intended to recursively update nested fields of the `user` object based on the user input `data`. However, if the `data` is attacker-controlled, it can be crafted to access unintended objects by traversing Python's built-in attributes, potentially causing severe consequences. At the top of Figure 1 is one possible exploit. It first uses the key `__class__` to retrieve the class object of `user` via `getattr` (line 5), then sets its `__getattribute__` method to a non-callable string `"1337"` (line 7). Since Python implicitly invokes `__getattribute__` for all attribute accesses, this change triggers a runtime exception on any access to `User` instances, resulting in a denial-of-service (DoS).

2.2. Vulnerability Taxonomy

In this subsection, we present the first taxonomy of class pollution vulnerabilities along three aspects: (i) pollution primitives (how to resolve and modify objects), (ii) pollution targets (what objects could be affected), and (iii) consequences (what security impacts may follow). Together, these are keys to understanding class pollution vulnerabilities in terms of the triage, detection, and exploit development.

2.2.1. Pollution Primitives

Class pollution vulnerabilities consist of multiple object access steps followed by a final object assignment step, as illustrated in Figure 3 (a). Each step of object access or assignment is defined as a primitive, with object access treated as a "get" primitive and the final assignment as a "set". Then, each primitive consists of one or more atomic operations as shown in Figure 3 (b). We start from atomic "get" and "set" operations with measurements of how they are used in practice and then describe how these atomic operations form into primitives.

TABLE 2: Systemization of reflected set operations in Python from builtins and standard libraries. The header follows Table 1.

	Features			Prevalence		
Code	Type	Apply	Origin	# Inst.	# Pack.	Spec.
`obj.__dict__[name]=val`	A	*	B	437.8K	3.1K	[20]
`setattr(obj,name,val)`	A	*	B	214.3K	12K	[39]
`object.__setattr__(obj,name,val)`	A	*	B	11.4K	1.6K	[40]
`obj.__setattr__(name,val)`	A	*	B	10.4K	2.3K	[40]
`dict[key]=val`	I	M/S	B	7.7M	36.8K	[28]
`dict.update(key=val)`	I	M	B	687.8K	22.7K	[41]
`dict.setdefault(key, val)`	I	M	B	111.1K	8K	[32]
`dict.__setitem__(key, val)`	I	M/S	B	7.7K	2.2K	[42]
`operator.setitem(dict,key,val)`	I	M/S	O	27	12	[43]
`operator.__setitem__(dict,key,val)`	I	M/S	O	0	0	[44]
`exec(f"EXPR†", {"o":obj})`	*	*	B	90	47	[38]
`eval(f"EXPR‡", {"o":obj})`	*	*	B	16	7	[37]

Legend: A: Attribute; I: Item; O: Object; M: Mapping; S: Sequence; B: Builtin; O (under Origin column): Operator; EXPR†: Any operation above where only the key name and value are attacker-controlled and all other expressions are sanitized. EXPR‡: Same as EXPR†, but excludes assignment statements (i.e., the first and fifth rows).

Atomic "Get" Operations. Python supports two atomic

TABLE 3: Diverse targets in Python class pollution through indirect access mechanisms. Each row lists the target type, access mechanism, a description, and a code snippet showing the loading context. Attacker-controlled values are highlighted in red and bolded. The row marked with ➕ is newly discovered in this work.

Pollutable Target	**Access Mechanism**	**Examples**	
		ⓘ **Description**	✸ **Loading Context**

Class	Attribute Lookup	`C.v`, where class `C` is accessible; the attacker controls the `cls.v` and `self.v` when the attribute is not defined on the instance.	`class C:`
			`  def __init__(self): self.v;`
			`  @classmethod`
			`  def other(self): cls.v;`

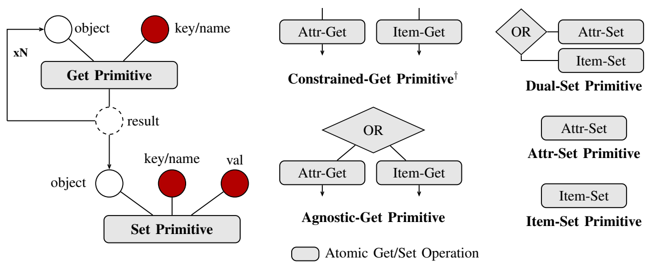
Module	Global Variable Reference	`mod.v`, where module `mod` is accessible; the attacker controls the global variable used across modules.	`# mod.py`
			`def f(): global v; v`
			`# other.py`
			`from mod import v; v`

Function	Global Variable Reference Module Lookup[1]	`f.__globals__["v"]`, where function `f` is accessible; the attacker controls the global variable `v` in its defining module.	`# mod.py` `# other.py`
			`def f(): pass` `import mod`
			`v` `mod.v`

	Local Variable Reference	`f.__kwdefaults__["p"]`, where function `f` is accessible; the attacker controls the default value of keyword-only parameter `p`.	`def f(*, p="default"): p`

➕ Function Closure	Local Variable Reference	`g.__closure__[i].cell_contents`, where `i` indexes the captured variable and function `g` is accessible; the attacker controls the captured variable `v`.	`def f(v):`
			`  def g(): v`
			`  return g`

[1] Module lookup here is an indirect access as the pollutable target is the function `f`, and `f.__globals__` is a dictionary reference—not the module itself.



(a) Pollution procedure. (b) Pollution atomics and primitives.

†: Item-Get atomic operation cannot appear as the first Get Primitive in (a).

Figure 3: Illustration of pollution primitives and atomics.

"get" operations that differ in their working namespace and reachability. First, attribute access, such as `getattr(obj, key)`, retrieves values from the attribute namespace and is supported by all Python objects. Second, item access, such as `dict[key]`, retrieves elements using keys or indexes and is only supported by container objects such as dictionaries, lists, and sets. Note that attribute access and item access operate on different namespaces—namely, attribute namespace and item namespace. Therefore, the two access operations are not interchangeable: an attribute cannot be accessed using item lookup, and vice versa.

The atomic "get" operation can appear in different syntactic forms in Python programs. To study these forms systematically, we analyzed the Python Standard Library [45] and identified 20 syntactic variants of "get" operations, as shown in Table 1. We group the operations based on whether they support attribute access, item access, or both. Each operation is marked with its access type, applicable object types, origin, and capabilities, including support for retrieving dunder attributes, methods, or other fields. To measure the usage of these operations in practice, we applied CodeQL [8] to statically analyze the 50,000 most-downloaded PyPI packages. We show the prevalence of these operations regarding their occurrences (i.e., individual uses) in Python applications and packages, respectively.

We further measure how the object and key are combined during access, denoted as first-order or second-order operations, in the Column "Order" in Table 1. In first-order operations, the object and key appear in the same expression and return the value in a single step (e.g., #1 and #11). In contrast, second-order operations split the object and key across expressions and require two steps to retrieve the value (e.g., #2 and #17). These second-order operations are more challenging to track in analysis, as they may introduce intermediate data flows between the two steps.

Atomic "Set" Operations. There are also two types of atomic "set" operations in Python, one setting attributes (*Attr-Set*) and the other setting items (*Item-Set*). Different from the atomic "get", the two atomic "set"s can be interchangeable when the object `obj` has a writeable `__dict__` attribute. For example, `obj.x=v` is semantically equivalent to `obj.__dict__["x"]=v`. We also performed a measurement study of Python "set" operations and identified twelve syntactically different "set" operations from the Python Standard Library [45] with a quantification of their real-world prevalence. Table 2 presents a summary of various "set" operations, grouped based on whether they support attribute assignment, item assignment, or both.

"Get" and "Set" Primitives. First, the left of Figure 3 (b) shows two types of "get" primitives: *Agnostic-Get* and *Constrained-Get*, distinguished by whether the attacker can freely choose the "get" atomic operations at each access step. An *Agnostic-Get* primitive allows an attacker to choose between `getattr` and `getitem` freely, which can be formed in two ways: (i) through a control-flow branch that allows the selection of different atomic gets across paths, or (ii) dynamically via reflection functions such as `eval` and `exec`, as shown in entries #19 and #20 of Table 1. In contrast, a *Constrained-Get* primitive requires the attacker to follow a fixed access pattern imposed by the program logic, e.g., a chain of attribute access only. Note that item access alone cannot form a valid *Constrained-Get* primitive for class pollution, since it operates within container objects like dictionaries and cannot affect shared runtime objects outside the container.

Second, the right of Figure 3 (b) shows three types of "set" primitives based on the attacker's ability to set the final object. In the *Dual-Set* primitive, the attacker can choose between attribute and item assignment. In the *Attr-Set* case, the attacker is restricted to attribute-based writes (e.g., `setattr(obj, key, v)`). In the *Item-Set* case, they are restricted to item-based writes (e.g., `obj[key] = v`). These distinctions parallel the two get primitive types seen in "get" operations and affect which fields or internal structures the attacker can overwrite.

Primitive Combinations. The three types of "set" primitives combined with the two types of "get" primitives define six types of class pollution vulnerabilities, each reflecting a distinct attacker capability in resolving and modifying runtime structures. To the best of our knowledge, only the combination of *Dual-Set* and *Agnostic-Get* is known to the community with existing vulnerability reports, and the rest five are newly defined and discovered in this paper.

2.2.2. Pollution Targets

We define a pollution target as a runtime object which is reachable via attribute or item access and can affect the program behavior. This excludes values that are inaccessible via such lookups or have no effect on program behavior, such as most local variables and type annotations. Understanding pollution targets helps to identify which objects can be polluted and how such pollution can affect program execution.

After the class pollution happens, pollution targets can be accessed by the program in two ways: (i) direct access, which occurs when the attacker modifies the exact attribute later used (e.g., setting `a.b` and using `a.b` later), and (ii) indirect access, which arises from Python's data model and reflective mechanism, e.g., modifying a class variable `v` affects the default values of `cls.v` and all instances' `self.v` values. Polluting a function's `__kwdefaults__` can change the interpretation of its keyword-only arguments [46], which are parameters defined after a "`*`" in the function signature and accepted only by name, e.g., the `arg1` in `f(*, arg1)`.

To systematically capture these possibilities, we analyzed Python's type system based on the built-in attributes defined in the Python data model [47]. Our analysis covers 39 data types and focuses on those that influence the program through indirect access mechanisms. Table 3 summarizes these pollution targets, their access mechanisms, and how attacker-controlled values can be implicitly propagated to affect program behavior. In total, there are four categories of pollution targets: (i) classes, (ii) modules, (iii) functions, and (iv) function closures. These targets can be indirectly accessed through different mechanisms, including attribute access via the `self` keyword, global variable references, and local variable references exposed through default parameters or closures.

2.2.3. Consequences

The consequence of class pollution is that an adversary corrupts program behaviors by changing either data or control flow via exploiting class pollution, leading to other well-known vulnerabilities. We list some possibilities and the associated consequential vulnerabilities below:

- Sink value corruption. The polluted value flows directly into a traditional vulnerability sink, e.g., `os.system`, `file.write`, and `requests.get`. The resulting impact includes remote code execution (RCE), arbitrary file writing, or server-side request forgery (SSRF).
- Call target corruption. An attacker can overwrite a defined callable with a non-callable primitive such as a string. When the callable is invoked, the program raises an exception that, if unhandled, can crash the application, leading to a Denial of Service.
- Corruption of the security condition. The attacker can change the outcome of a security-sensitive condition, enabling access to functionality that would otherwise be unreachable. For instance, polluting a secret key may bypass authentication checks, and overwriting a sanitizer may allow untrusted input to bypass validation.

2.3. Threat Model

In this subsection, we describe the threat model of Pyrl. The victim in our threat model is a vulnerable Python package that processes different sources of attacker-controlled input as described below:

- Remote Input. Such inputs are from network interfaces, such as HTTP requests, WebSocket connections, or remote procedure calls (RPCs). For example, consider a web server that processes requests from remote clients. An adversary can send a crafted request containing malicious input to exploit a vulnerability in the server-side logic.
- Local Input. Such inputs are from local resources, such as command-line arguments and data files. They are considered controllable by adversaries because (i) untrusted data files could be downloaded from attacker-controlled sources, e.g., an external URL or a malicious commit or pull request on GitHub, and (ii) command-line arguments may be from large language models (LLMs) through prompt injection for an LLM agent or when the application is locally invoked via the Model Context Protocol (MCP).
- Package-level Input. Such package-level inputs accept data from another package or application, which may be

controlled by an adversary either remotely or locally. Vulnerable packages with such inputs are not independently exploitable, because they need to be exported with another application so that adversary's inputs can reach the vulnerability location. Prior works—including academics [48]–[55] and industry [56]–[59]—all consider package-level inputs as potentially controlled by adversaries.

3. Overview

We start from a motivating example in §3.1 and then describe the detection challenge and an overview of our solution in §3.2.

3.1. A Motivating Example

Listing 1 presents a real-world zero-day class pollution vulnerability (CVE-2025-24370) discovered by Pyrl in django-unicorn [60], a popular reactive web framework for Django. The vulnerability lies in the core server-side logic responsible for handling client-side data updates. It allows attackers to pollute any reachable server-side objects by traversing the component instance hierarchy through crafted client-side inputs. We have responsibly reported the vulnerability to the developers, who have since fixed it.

Vulnerability Details. The `message` function (line 2) handles POST requests for updating server-side components based on client-side input. For each `action` in the `action_queue` from the request (line 6), the code extracts a `name` (line 7) and `value` (line 8) pair from the payload, then passes them to the `set_property_value` function (line 10) to perform the update.

The core update logic lies in `set_property_value` function (line 12-24). The function first splits the client-provided `name_str` into an array of dot-separated `names` (line 13), and then use them to iteratively resolve the component object via reflective access operations. At each iteration (lines 15–24), the function checks whether the current object has the specified attribute using `hasattr` (line 15), or whether it is a dictionary (line 20), and then selects the appropriate access operation: attribute access with `getattr` (line 19) or item access with square brackets (line 24). In the final iteration, the last `name` is reached, the value is updated using either attribute assignment (line 17) or item assignment (line 22), depending on the object's type.

While the handler is intended to conveniently set deeply nested attributes or items within the `component` object, it can be exploited by an attacker to overwrite values in unexpected runtime objects using carefully-crafted dot-separated names. Since the lookup path is derived directly from untrusted client input and applied without any validation, an attacker can construct a malicious name like `__class__.__init__.__globals__.v` to overwrite the value `v` in the global scope of the module where the `component` class is defined. The attacker may even traverse further and modify values in other modules, leading to severe security consequences.

Exploitation and Impact. We describe four different consequences that this class pollution in django-unicorn can lead to. Note that the details of the exploit code can be found in Appendix A (Table 9).

```
1  @require_POST
2  def message(request: HttpRequest, comp_name=None):
3    comp_req = ComponentRequest(request, comp_name)
4    component = View.create(comp_req.id, comp_req.name,
       ↪ request)
5
6    for action in comp_req.action_queue:
7      prop_name = action.payload.get("name")
8      prop_value = action.payload.get("value")
9
10     set_property_value(component, prop_name, prop_value
       ↪ )
11
12 def set_property_value(component, name_str, value):
13   names = name_str.split(".")
14   for idx, name in enumerate(names):
15     if hasattr(component, name):              /*Braching*/
16       if idx == len(names) - 1:
17         setattr(component, name, value)       /*Sink #1*/
18       else:
19         component = getattr(component, name)
20     elif isinstance(component, dict):
21       if idx == len(names) - 1:
22         component[name] = value               /*Sink #2*/
23       else:
24         component = component[name]
```

Listing 1: A motivating example of a zero-day class pollution vulnerability found in `django-unicorn` [60]. The code is simplified for explanation.

- **DoS.** Overwriting frequently used class methods like `__getattribute__` with a non-callable value, such as a string, would cause any attribute access on the affected object to raise an exception, leading to a denial-of-service.
- **Stored XSS.** By default, django-unicorn escapes user input to prevent any reflected XSS, using an entity encoding function from the BeautifulSoup library [61]. This function relies on a character-to-entity mapping (e.g., < to `&lt`). By overwriting all entries in the map with an XSS payload, any user input would be rendered with the attacker's injected script. We refer to this as universal stored XSS, as it affects all website users and is not limited to a specific page.
- **Authentication Bypass.** The application adopts the Django framework's secret key to sign and verify session cookies. This key is stored as a global variable in the `settings` module. We found that the `settings` module is reachable from the base object, allowing an attacker to overwrite the secret key with a known value. This enables forging server-signed content, leading to authentication bypass.
- **RCE.** Component names and their corresponding class locations are cached by the application and later used in Python's `import` calls. Our analysis revealed that this cache can be polluted to load arbitrary modules. Specifically, importing the standard `antigravity` module triggers a browser launch using the `BROWSER` environment variable. By overwriting

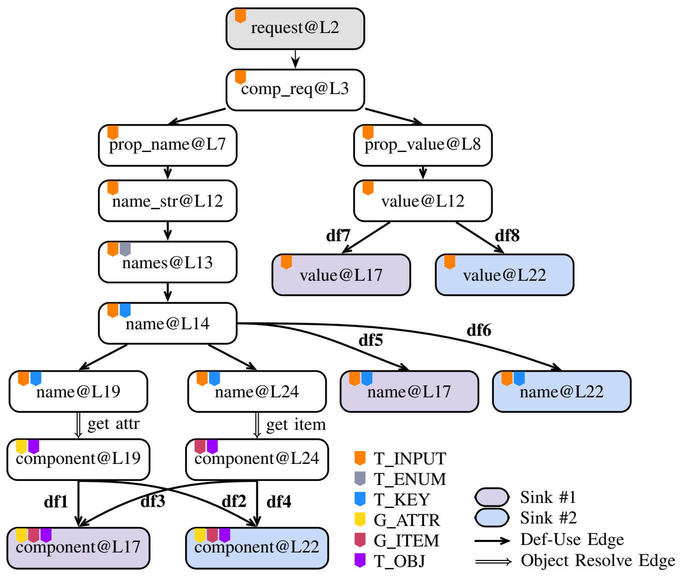


Figure 4: Taint Flow Graph of Listing 1. Pyrl detects eight taint flows (df1–df8) in this case. Each node is annotated with semantic taint labels at its top-left corner. Two sets of sinks are highlighted in different colors.

`sys.environ.BROWSER` with an arbitrary shell command, we achieve remote code execution.

3.2. Challenge and Our Solution

We now describe the key challenge in detecting the vulnerability and our solution. As seen in the motivating example in Listing 1, the "get" and "set" primitives reside in branch statements within a loop where the iterable is attacker-controlled. This allows for repeated execution of the loop and arbitrary selection of different forms of "get" and "set" on each iteration, which greatly increases the complexity of flow tracking.

Pyrl solves the challenge using fine-grained, expressive labels that track from attackers' input to sinks through "get" and "set" primitives. It assigns a label `T_INPUT` to direct attacker input, and derives `T_ENUM` and `T_KEY` at the iterable generation and the loop variable iteration, respectively. During object resolution, where "get" primitives are used, Pyrl assigns a `T_OBJ` label to the resulting object if its key is tainted, and uses `G_ATTR` and `G_ITEM` to distinguish between attribute access and item access. To model "get" primitives across different branches, Pyrl merges the labels from disjoint control-flow paths at the end of branch statements. Finally, at the assignment, Pyrl checks the taint labels of the assignment tuple to verify and classify the vulnerability.

Now we illustrate the operational taint propagation in the motivating example with Listing 1 and Figure 4. The taint originates from the attacker's input at line 2 in Listing 1, denoted as `request@L2` with a label `T_INPUT` in Figure 4. When this `T_INPUT` is split into an iterable sequence, the resulting sequence is labeled as `T_ENUM` (`names@L13`), and each iterated element is labeled as a `T_KEY` (`name@L14`). Subsequently, when a "get" primitive occurs, for example, attribute access where the key is tainted (line 19), the resolved object is labeled `T_OBJ` and `G_ATTR` to reflect the result and its access type (`component@L19`). At the end of each iteration, Pyrl merges the labels from branches through a ϕ node. For example, the variable `component` on lines 19 receives the merged label `T_OBJ`, which comes from the result of attribute access (line 19) and item access (line 24), indicating that it can be resolved through the "get" primitive *Agnostic-Get*.

Finally, when Pyrl encounters a "set" primitive, such as line 17 and line 22 in Listing 1, it checks the taint of its assignment tuple. Pyrl considers a "set" to be a valid sink only when the object is labeled with `T_OBJ`, the key labeled with `T_KEY` or uncompounded `T_INPUT`, and the value with `T_INPUT`. Besides, Pyrl classifies the "set" primitive in this example as *Dual-Set* because the objects on line 17 and line 22 are fetched from the same ϕ node. Combined with the *Agnostic-Get* for the "get" primitive, Pyrl classifies the case as *Agnostic-Get*$\times$*Dual-Set*.

4. Design

We start from the operational taint analysis in §4.1, and then elaborate on the vulnerability specification in §4.2.

4.1. Operational Taint Analysis

In this subsection, we first describe the target language and then detail the taint initialization and propagation rules.

4.1.1. Language and Notation

For clarity, we describe the operational semantics for a core subset of Python constructs relevant to taint tracking. This includes assignments, control-flow structures (e.g., conditions and loops), and expressions with variable references, constants, binary operations, item and attribute access, attribute assignments, and function calls.

Next, we present the notations used in the operational semantics of the taint analysis in Table 4. A program state σ is represented as a tuple $(\Sigma, \tau_\Delta, pc, \iota)$, where Σ maps a statement number to its corresponding statement. The taint map τ_Δ associates each variable or memory location with a set of taint labels. We denote the taint update of variable x to label l as $\tau_\Delta[x \mapsto \{l\}]$, and the removal of label l from x as $\tau_\Delta[x \mapsto \tau_\Delta[x] \setminus l]$. The check of a value holds a specific label l is denoted as $l \in \tau_\Delta[x]$. We further define operational semantics rules for taint initialization and propagation in Figure 5, attached in the appendix.

4.1.2. Taint Initialization

In Python class pollution, the initial taint originates from attacker-controlled input, which Pyrl models as an *input* call. During evaluation, any call to *input* produces a new value labeled with `T_INPUT`$^\alpha$ where α denotes the input location.

The INPUT rule abstraction covers three types of channels that receive external input in Python programs. These include: (i) remote inputs in web applications, such as data received through network interfaces from standard libraries

TABLE 4: Notations used in Pyrl, including execution context variables and semantic taint labels.

Context	Description
Σ	Maps a statement number to a statement.
$\tau_{\triangle}$	Maps each variable or memory location to its semantic labels.
pc	The program counter, indicating the instruction being executed.
ι	The next instruction to be executed.
Label l	**Description**
`T_INPUT`$^{\alpha}$	Value directly controlled by the attacker from origin location α.
`T_ENUM`	Tainted enumerable value from split operation.
`T_KEY`	Potential key value derived from enumeration.
`T_OBJ`	Object resolved through a tainted key.
`G_ATTR`$^{\alpha}$	Resolved via attribute access (e.g., `getattr(obj, key)`) from origin location α.
`G_ITEM`$^{\alpha}$	Resolved via item access (e.g., `obj[key]`) from origin location α.
`NON_TAINT`	Value that is constant literal or unaffected by attacker input.
Operator	**Description**
$\uplus$	Disjunction of get labels from different control-flow paths.
$\oplus$	Composition of a value from both tainted and non-tainted parts.

like http.server and socket, as well as widely used third-party frameworks like Flask and Django; (ii) local inputs in local applications. Examples are command-line arguments or data read from files; and (iii) package-level inputs in third-party libraries, which consist of exported functions and class methods that may be invoked by other code. Pyrl treats functions and class methods exposed at the package level as potential entry points for attackers. Names starting with an underscore are excluded, as they are conventionally treated as internal and not part of the public interface.

4.1.3. Taint Propagation via Semantic Labels

Taint propagation via semantic labels can happen in four cases, depending on how the label evolves during propagation: (i) label derivation, where a new taint label is derived based on the original label, and both labels are retained; (ii) label transition, where the taint label changes from one type to another (e.g., from `T_ENUM` to `T_KEY`); (iii) label retention, where the same taint label is passed along unchanged to the new variable, as those used in traditional taint analysis; and (iv) label merging, where multiple labels from different control-flow paths are joined. Each case involves a specific label transformation rule, which we describe below.

Label Derivation. Label derivation occurs when a variable labeled `T_INPUT` is transformed into an iterable or loop variable, either through a string split or iterable generation. During label derivation, the original `T_INPUT` label is retained alongside the derived label. This enables nested transformations: if another split is later applied to the result, the same derivation logic continues to apply. The key label transitions are as follows:

- `T_INPUT` $\rightarrow$ `T_ENUM`. A value returned by a *split* operation is labeled `T_ENUM` if the base object is labeled `T_INPUT`, as defined by the rule SPLIT. The label `T_ENUM` captures the fact that the attacker-controlled input has become an enumerable sequence. Class pollution requires iterating over distinct attacker-controlled keys. In contrast, if the same key—although under attacker control—is reused across multiple get operations, it cannot traverse different objects and would not result in class pollution. By assigning `T_ENUM`, Pyrl marks the input as capable of yielding a sequence of distinct keys when enumerated.
- `T_INPUT` $\rightarrow$ `T_KEY`. In addition to transitions via `T_ENUM`, elements obtained by iterating over a `T_INPUT` value may be labeled `T_KEY` directly, as defined by rule ENUMERATE. This accounts for cases where the input is already an enumerable structure, such as a list or dictionary parsed from formats like JSON or YAML.

```
1 def foo(obj, path):
2 + path = "abc." + path                        variant ❶
3   keys = path.split(".")
4 + keys.append(".abc")                         variant ❷
5   for key in keys:
6     obj = getattr(obj, key)
7   return obj
```

Listing 2: Variants of taint propagation from `T_INPUT` to `T_ENUM` under partial control. Lines 2 and 4 illustrate the effect of prepending and appending non-tainted values, respectively, which influences how taint is propagated during bounded loop unrolling.

Label Transition. A label may transition to another when the resulting value no longer holds the original property. The key label transitions are:

- `T_ENUM` $\rightarrow$ `T_KEY`. When elements are accessed from a `T_ENUM` value, either through enumeration or item access, each resulting element is labeled as `T_KEY`, as defined by rules ENUMERATE and GETITEM. The label `T_KEY` captures the use of individual elements derived from an attacker-controlled sequence, indicating that they may be used as keys in subsequent get operations.
  We discuss the cases where the `T_ENUM` value is not fully controlled by the attacker. Pyrl adopts an approximation over prefix- or suffix-based taint mixing by distinguishing between `NON_TAINT`$\oplus$`T_ENUM` and `T_ENUM`$\oplus$`NON_TAINT`.
  Consider the example in code snippet 2, where the original keys are fully attacker-controlled. Variants ❶ and ❷ show cases where non-attacker-controlled values are prepended or appended, respectively. Pyrl performs bounded loop unrolling with a fixed count (two rounds), following the practice of bounded static analysis [62]. To reflect partial control, if the label is `NON_TAINT`$\oplus$`T_ENUM`, Pyrl only propagates the taint in the second round. If the label is `T_ENUM`$\oplus$`NON_TAINT`, taint is propagated in the first round only. This approximation is tailored to match the nature of class pollution, where early keys not under attacker control do not block exploitation, as long as later keys are attacker-controlled. Therefore, in variant ❶, the object obj still receives the `T_OBJ` label in the last iteration. In contrast, in variant ❷, where the final key is non-tainted,

$$\frac{\sigma{=}(\Sigma,\tau_{\Delta},pc,\iota),\ x_{ret}{=}input()^{\alpha},\ t'{=}\texttt{T\_INPUT}^{\alpha}}{\langle x{\leftarrow}input()^{\alpha},\sigma\rangle{\Rightarrow}\langle\sigma'{=}(\Sigma,\tau_{\Delta}[x{\mapsto}t'],pc{+}1,\iota)\rangle}\ \textsc{Input} \quad \frac{\sigma,\ x_{ret}{=}split(E,sep),\ t{=}\tau_{\Delta}[E],\ t'{=}\{\texttt{T\_ENUM}\}{\cup}t\ \ if\ \ \texttt{T\_INPUT}^{\alpha}{\in}t\ \ else\ \ t'{=}t}{\langle x{\leftarrow}split(E,sep),\sigma\rangle{\Rightarrow}\langle\sigma'{=}(\Sigma,\tau_{\Delta}[x{\mapsto}t'],pc{+}1,\iota)\rangle}\ \textsc{Split}$$

$$\frac{\sigma{=}(\Sigma,\tau_{\Delta},pc,\iota),\ x_{ret}{=}enum(E),\ t{=}\tau_{\Delta}[E],\ t'{=}\{\texttt{T\_KEY}\}{\cup}\{t{\setminus}\texttt{T\_ENUM}\}\ \ if\ \ \texttt{T\_INPUT}^{\alpha}{\in}\ t\ \ else\ \ t'{=}t}{\langle x{\leftarrow}enum(E),\sigma\rangle{\Rightarrow}\langle\sigma'{=}(\Sigma,\tau_{\Delta}[x{\mapsto}t'],pc{+}1,\iota)\rangle}\ \textsc{Enumerate}$$

$$\frac{\sigma{=}(\Sigma,\tau_{\Delta},pc,\iota),\ x_{ret}{=}o[k],\ t'{=}\{\texttt{G\_ITEM}^{\alpha},\texttt{T\_OBJ}\}\ \ if\ \ \texttt{T\_KEY}{\in}\tau_{\Delta}[k]\ \ and\ \ \texttt{T\_OBJ}{\in}\tau_{\Delta}[o]\ \ else\ \ t'{=}t{\oplus}\texttt{NON\_TAINT}}{\langle x{\leftarrow}o[k],\sigma\rangle{\Rightarrow}\langle\sigma'{=}(\Sigma,\tau_{\Delta}[x{\mapsto}t'],pc{+}1,\iota)\rangle}\ \textsc{GetItem}$$

$$\frac{\sigma{=}(\Sigma,\tau_{\Delta},pc,\iota),\ x_{ret}{=}getattr(o,k),t'{=}\{\texttt{G\_ATTR}^{\alpha},\texttt{T\_OBJ}\}\ \ if\ \ \texttt{T\_KEY}{\in}\tau_{\Delta}[k]\ \ else\ \ t'{=}t{\oplus}\texttt{NON\_TAINT}}{\langle x{\leftarrow}getattr(o,k),\sigma\rangle{\Rightarrow}\langle\sigma'{=}(\Sigma,\tau_{\Delta}[x{\mapsto}\{\texttt{G\_ATTR}^{\alpha},\texttt{T\_OBJ}\}],pc{+}1,\iota)\rangle}\ \textsc{GetAttr}$$

$$\frac{\sigma{=}(\Sigma,\tau_{\Delta},pc,\iota),\ \phi(x_1,x_2),\ t_1{=}\tau_{\Delta}[x_1],\ t_2{=}\tau_{\Delta}[x_2],\ t'{=}\texttt{G\_ATTR}^{\alpha1}{\uplus}\texttt{G\_ITEM}^{\alpha2}\ \ if\ \ \{\texttt{T\_OBJ},\texttt{G\_ATTR}^{\alpha1}\}{\subseteq}t_1\ \ and\ \ \{\texttt{T\_OBJ},\texttt{G\_ITEM}^{\alpha2}\}{\subseteq}t_2}{\langle x{\leftarrow}\phi(x_1,x_2),\sigma\rangle{\Rightarrow}\langle\sigma'{=}(\Sigma,\tau_{\Delta}[x{\mapsto}t'],pc{+}1,\iota)\rangle}\ \textsc{Branch}$$

Figure 5: Operational semantics of Pyrl, illustrating taint label initialization, propagation, transition, and merging rules.

the object will not receive T_OBJ, as the taint does not propagate to the final iteration.

- T_KEY $\rightarrow$ T_OBJ. When a key labeled T_KEY is used in a get operation, the resulting value may be labeled with T_OBJ to indicate that it is an object resolved by the attacker. Pyrl differentiates the access types with sub-labels: G_ATTR$^{\alpha}$ and G_ITEM$^{\alpha}$, where α denotes the access location.

  For attribute access, as defined by rule GETATTR, if the key is labeled T_KEY, the result is labeled T_OBJ and annotated with G_ATTR$^{\alpha}$ to reflect the access mechanism. For item access, as defined by rule GETITEM, the behavior depends on whether the base object is already labeled T_OBJ. If so, the result inherits T_OBJ and receives the sub-label G_ITEM$^{\alpha}$. Otherwise, item access alone does not introduce the T_OBJ label as discussed in Section 2.2.1.

  Pyrl handles all the get operations listed in Table 1, including second-order access patterns. For example, in `operator.attrgetter(name)(obj)`, the key and base object appear in separate expressions. In such cases, Pyrl tracks additional data flow on the intermediate result of the first expression and defers propagation until the second expression is analyzed, at which point it applies the T_OBJ label and the corresponding sub-label based on the resolved access.

Label Retention. We refer to label retention as passing the same label from one variable to another. To avoid overtainting, Pyrl applies different propagation policies depending on which semantic labels a value holds.

- T_INPUT $\rightarrow$ T_INPUT. This label follows a relaxed propagation policy. It propagates through binary and unary operations, attribute and subscript access, enumeration, as well as built-in and standard library function calls like `string.toUpperCase`. The relaxed policy is due to the fact that even if an attacker controls only part of the value in the sink (e.g., a substring in a command line argument), it can still trigger injection-style vulnerabilities.
- T_ENUM $\rightarrow$ T_ENUM. This label retention happens during sequence-related operations, such as sorting and slicing. When sequences are updated in cases where the operation includes new non-attacker-controlled values, Pyrl distinguishes the position where the value is added (e.g., `append` vs. `insert`) and accordingly updates the taint to T_ENUM$\oplus$NON_TAINT or NON_TAINT$\oplus$T_ENUM.
- T_KEY $\rightarrow$ T_KEY. This label follows a strict propagation policy. Values labeled T_KEY serve as keys in attribute or item access operations, so Pyrl only allows propagation to string transformations that preserve their ability to resolve to the intended target. For instance, binary string operations (e.g., concatenation) propagate the label only when both operands are attacker-controlled. String methods such as `strip` also preserve the label when applied with whitespace characters. On the other hand, operations that introduce non-attacker-controlled values or manipulate the original value do not propagate the label, as the resulting string may no longer match the intended attribute or key.
- T_OBJ $\rightarrow$ T_OBJ. This label also follows a strict propagation policy. The label only propagates through value-preserving data-flow steps, including direct assignments, and indirect heap-based data flows, such as storing the value in a collection and retrieving it later. Note that Pyrl does not propagate the label through get operations, as the resulting value no longer refers to the attacker-resolved object.

Label Merging. Pyrl performs label merging at control-flow join points to distinguish between *Agnostic-Get* and *Constrained-Get* cases. Specifically, when merging two variables at a branching join, written as $\phi(x_1, x_2)$, if both hold T_OBJ—one labeled with G_ATTR$^{\alpha_1}$ and the other with G_ITEM$^{\alpha_2}$—Pyrl assigns G_ATTR$^{\alpha_1}$ $\uplus$ G_ITEM$^{\alpha_2}$ to the merged variable in τ_{Δ}. This disjunctive label indicates that either get primitives may be taken at runtime. Note that the disjunction operator ($\uplus$) is introduced only at control-flow join points. Sequential get operations, such as applying `getattr` followed by `getitem`, do not trigger label merging and are treated as a constrained access path. We show the label merging in the rule BRANCH. For other taint labels, such as T_INPUT, T_ENUM, and T_KEY, Pyrl updates τ_{Δ} directly during propagation to avoid overtainting.

4.2. Vulnerability Detection

In this subsection, we describe how Pyrl detects the six types of class pollution vulnerabilities defined in our taxonomy. We first specify the label conditions under which each type of vulnerability occurs, and then describe how Pyrl further checks their exploitability in practice.

4.2.1. Vulnerability Specification

Based on the taint analysis result, Pyrl checks for vulnerabilities by analyzing the taint labels associated with values at the sink—specifically, attribute assignment or item assignment, as summarized in Table 2. Each assignment is represented as a triplet (o, k, v), where o is the object being written to, k is the key or attribute being used, and v is the value being assigned. We use subscripts a and i to distinguish between attribute assignments, e.g., (o_a, k_a, v_a), and item assignments, e.g., (o_i, k_i, v_i).

Pyrl detects all types of class pollution vulnerability based on the specification in Table 5. To illustrate, consider the case of *Agnostic-Get×Dual-Set*, where the vulnerability allows the attacker to resolve the object through both get operations and set it through both set operations. First, the target objects o_i and o_a must both carry the label T_OBJ with the same sub-label $\texttt{G\_ITEM}^{\alpha_1} \uplus \texttt{G\_ATTR}^{\alpha_2}$. This disjunctive label indicates that the object may be resolved through either attribute access at α_1 or item access at α_2, depending on the control-flow path. Since T_OBJ only propagates through data flow, its presence on both o_i and o_a implies that they refer to the same runtime object. Next, the same attacker-controlled input must propagate to both the key and the value used in each assignment. Note that the key used in the attribute assignment must carry either the T_KEY label or an uncompounded T_INPUT label—that is, not composed with NON_TAINT—to ensure it is fully attacker-controlled.

4.2.2. Exploitability Checking

The mere existence of taint flows does not guarantee that the vulnerability is exploitable. In particular, for vulnerabilities involving two assignment sinks, i.e., the *Dual-Set* case, successful exploitation requires that the two assignments reside in mutually exclusive control-flow blocks, as discussed earlier. If both assignments are executed sequentially along any path, one may cause a runtime error (e.g., applying item access to a non-container object), preventing successful exploitation. Therefore, Pyrl checks that the two sink locations are placed in mutually exclusive branches, ensuring that they are never executed together along any control-flow path.

Moreover, while Pyrl avoids over-tainting and favors precision in taint propagation, certain control-flow conditions may prevent successful pollution. We refer to such conditions as barrier nodes. Pyrl considers two types of barrier nodes: (i) key barriers, such as checks for the presence of a leading underscore or dot; (ii) object barriers, such as type checks that restrict the object to specific types.

To address this, Pyrl performs a control-flow dominator analysis from the assignment back to the input source. A vulnerability is excluded if a barrier node dominates the assignment—that is, all paths leading to the assignment pass through the barrier. This is commonly seen when the assignment appears within a conditional block guarded by a type check on the target object.

TABLE 5: Vulnerability specification with corresponding taint label conditions.

Type	Condition
Agnostic-Get×Dual-Set	$\{\texttt{T\_OBJ}, \texttt{G\_ATTR}^{\alpha 1} \uplus \texttt{G\_ITEM}^{\alpha 2}\} \subseteq \tau_\triangle[o_a] \cap \tau_\triangle[o_i]$ $\wedge\ \mathsf{ValidKey}^\dagger(\tau_\triangle[k_a]) \wedge \mathsf{ValidKey}(\tau_\triangle[k_i])$ $\wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_a] \wedge \texttt{T\_INPUT} \in \tau_\triangle[v_i]$
Constrained-Get×Dual-Set	$\texttt{T\_OBJ} \in \tau_\triangle[o_a] \cap \tau_\triangle[o_i]$ $\wedge\ \{\texttt{G}_*^\alpha\} \cap \tau_\triangle[o_a] = \{\texttt{G}_*^\alpha\} \cap \tau_\triangle[o_i]$ $\wedge\ \texttt{G\_ATTR}^{\alpha 1} \uplus \texttt{G\_ITEM}^{\alpha 2} \notin \tau_\triangle[o_a] \cap \tau_\triangle[o_i]$ $\wedge\ \mathsf{ValidKey}(\tau_\triangle[k_a]) \wedge \mathsf{ValidKey}(\tau_\triangle[k_i])$ $\wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_a] \wedge \texttt{T\_INPUT} \in \tau_\triangle[v_i]$
Agnostic-Get×Attr-Set	$\{\texttt{T\_OBJ}, \texttt{G\_ATTR}^{\alpha 1} \uplus \texttt{G\_ITEM}^{\alpha 2}\} \subseteq \tau_\triangle[o_a]$ $\wedge\ \mathsf{ValidKey}(\tau_\triangle[k_a])$ $\wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_a]$ $\wedge\ \nexists(o_i, k_i, v_i)$ s.t. $\{\texttt{T\_OBJ}, \texttt{G\_ATTR}^{\alpha 1} \uplus \texttt{G\_ITEM}^{\alpha 2}\} \subseteq \tau_\triangle[o_i]$ $\wedge\ \mathsf{ValidKey}(\tau_\triangle[k_i]) \wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_i]$
Constrained-Get×Attr-Set	$\texttt{T\_OBJ} \in \tau_\triangle[o_a] \wedge \texttt{G\_ATTR}^{\alpha 1} \uplus \texttt{G\_ITEM}^{\alpha 2} \notin \tau_\triangle[o_a]$ $\wedge\ \mathsf{ValidKey}(\tau_\triangle[k_a])$ $\wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_a]$ $\wedge\ \nexists(o_i, k_i, v_i)$ s.t. $\{\texttt{G}_*^\alpha\} \cap \tau_\triangle[o_a] = \{\texttt{G}_*^\alpha\} \cap \tau_\triangle[o_i]$ $\wedge\ \mathsf{ValidKey}(\tau_\triangle[k_i]) \wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_i]$
Agnostic-Get×Item-Set	$\{\texttt{T\_OBJ}, \texttt{G\_ATTR}^{\alpha 1} \uplus \texttt{G\_ITEM}^{\alpha 2}\} \subseteq \tau_\triangle[o_i]$ $\wedge\ \mathsf{ValidKey}(\tau_\triangle[k_i])$ $\wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_i]$ $\wedge\ \nexists(o_a, k_a, v_a)$ s.t. $\{\texttt{T\_OBJ}, \texttt{G\_ATTR}^{\alpha 1} \uplus \texttt{G\_ITEM}^{\alpha 2}\} \subseteq \tau_\triangle[o_a]$ $\wedge\ \mathsf{ValidKey}(\tau_\triangle[k_a]) \wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_a]$
Constrained-Get×Item-Set	$\texttt{T\_OBJ} \in \tau_\triangle[o_i] \wedge \texttt{G\_ATTR}^{\alpha 1} \uplus \texttt{G\_ITEM}^{\alpha 2} \notin \tau_\triangle[o_i]$ $\wedge\ \mathsf{ValidKey}(\tau_\triangle[k_i])$ $\wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_i]$ $\wedge\ \nexists(o_a, k_a, v_a)$ s.t. $\{\texttt{G}_*^\alpha\} \cap \tau_\triangle[o_a] = \{\texttt{G}_*^\alpha\} \cap \tau_\triangle[o_i]$ $\wedge\ \mathsf{ValidKey}(\tau_\triangle[k_a]) \wedge\ \texttt{T\_INPUT} \in \tau_\triangle[v_a]$

†: $\mathsf{ValidKey}(\tau_\triangle[k]) \triangleq \texttt{T\_KEY} \in \tau_\triangle[k]\ \vee\ (\texttt{T\_INPUT} \in \tau_\triangle[k] \wedge \texttt{NON\_TAINT} \notin \tau_\triangle[k])$

5. Implementation

We implemented Pyrl in the QL language, running on CodeQL [8] v2.21.3 with Python language support v4.0.5. CodeQL is a framework that models programs as a relational database and provides basic static analyses over syntax, control-flow, and data-flow facts. Analyses are written in QL language and can implement custom taint analysis over these facts. Our implementation consists of 3,509 lines of newly written QL code. Notably, CodeQL and its standard query suite do not support class pollution vulnerability detection. We first modeled all the necessary expressions and API calls used in our operational taint analysis as taint flow steps. Then, we leveraged the global taint tracking module with flow-state support to track semantic labels and perform the taint analysis. For exploitability checking, we implemented our analysis based on the control flow graph and call graph constructed by CodeQL. In addition to the analysis code, we extended the CodeQL standard library to support data flow steps through data structures provided by the collection library, e.g., `namedtuple`, and higher-order functions such as `reduce`. We also improved the default points-to analysis by enhancing the resolution of object attribute definitions. Finally, we developed a workflow to automate the analysis at scale, including package downloading, database setup, query execution, and result processing. We open source Pyrl, together with the workflow at the repository

TABLE 6: [RQ1] A selective list of zero-day vulnerabilities detected by Pyrl. Vulnerabilities triggered via Remote and Local inputs include their exploitation consequences. Package entries indicate vulnerabilities triggered through downstream applications, where exploitability is not directly observable.

Application	Popularity	Version	Input	Get Primitive	Set Primitive	Consequence	Status	CVE Identifier
				Constrained/Agnostic	Attr/Item/Dual			
GitHub Repositories (Sorted by # of Stars)								
ComfyUI	78.5K	v0.3.39	Remote	Constrained	Attr	DoS	Fixed	CVE-2025-6107
RAGFlow	52.8K	v0.19.0	Remote	Constrained	Attr	DoS	Reported	-
Taipy	18.1K	v4.0.3	Remote	Constrained	Attr	DoS,XSS,RCE,TL	Fixed	CVE-2025-30374
sd-webui-controlnet	17.6K	v1.1.436	Remote	Constrained	Attr	DoS	Reported	-
stable-diffusion-webui-forge	10.9K	latest	Remote	Constrained	Attr	DoS	Reported	-
Mesop	6.3K	v0.14.0	Remote	Constrained	Dual	DoS,RE	Fixed	CVE-2025-30358
Azure CLI	4.2K	v2.68.0	Local	Agnostic	Dual	TL,OSCI	Fixed	CVE-2025-24049
docarray	3.1K	v0.40.1	Remote	Constrained	Attr	DoS	Reported	CVE-2025-5150
django-unicorn	2.5K	v0.62.0	Remote	Agnostic	Dual	DoS,XSS,RCE,AB	Fixed	CVE-2025-24370
sverchok	2.3K	v1.3.0	Local	Agnostic	Dual	TL	Reported	CVE-2025-3982
fastapi-amis-admin	1.3K	v0.7.3	Remote	Constrained	Attr	DoS	Reported	-
PyPI Packages (Sorted by # of Weekly Downloads)								
accelerate	2.7M	v1.7.0	Package	Constrained	Attr	-	Reported	-
spaCy	2.6M	v3.8.7	Package	Constrained	Attr	-	Reported	-
magicattr	1.5M	v0.1.6	Package	Agnostic	Dual	-	Reported	-
glom	1.4M	v24.11.0	Package	Agnostic	Dual	-	Acknowledged	-
google-generativeai	1.3M	v0.8.5	Package	Constrained	Attr	-	Reported	-
diffusers	86.2K	v0.33.1	Package	Constrained	Attr	-	Reported	-
tf-keras	59.5K	v2.19.0	Package	Agnostic	Dual	-	Reported	-
mo-dots	55.2K	v10.678.x	Package	Agnostic	Dual	-	Reported	-
pykka	15.7K	v4.2.0	Package	Constrained	Attr	-	Reported	-

TL: Token Leakage; RE: Role Escalation (impersonate assistant/system roles in LLMs); AB: Authentication Bypass; OSCI: OS Command Injection;

(https://github.com/jackfromeast/python-class-pollution.)

6. Evaluation

In this section, we structure our evaluation of Pyrl around the following Research Questions (RQs).

- **RQ1** [Zero-day]: How many zero-day class pollution vulnerabilities does Pyrl detect? What are their types and prevalence?
- **RQ2** [FPs&FNs]: What are the false positives (FPs) and false negatives (FNs) of Pyrl?
- **RQ3** [Ablation Study]: How does operational taint analysis reduce false positives compared to traditional taint analysis in detecting class pollution vulnerabilities?
- **RQ4** [Case Studies]: What do real-world class pollution vulnerabilities look like, and how does Pyrl detect them?
- **RQ5** [Performance]: What is the performance of Pyrl? How well does it scale when analyzing large code bases?

6.1. RQ1: Zero-day Vulnerabilities

In this subsection, we answer the research question regarding the zero-day vulnerabilities detected by Pyrl. We define a detected vulnerability as a zero-day if there is no prior public disclosure and it is confirmed by a human expert with a successful exploit. Note that Pyrl treats multiple taint flows that reach the same sink as a single vulnerability, rather than counting each flow separately. This is based on the observation that, in real-world vulnerability reports, issues with the same sink are typically addressed by a single patch and assigned one CVE identifier.

Datasets. We evaluated Pyrl on two large-scale datasets: (i) GitHub repositories containing Python code with over 100 stars [18]. We crawled the dataset in January 2025, resulting in a total of 69,361 repositories. (ii) all available packages from PyPI [1], which is the official Python package repository. We crawled the dataset in March 2025, collecting a total of 602,114 packages. Together, these datasets cover over half a million real-world Python programs.

Results. In total, Pyrl reported 868 vulnerabilities, among which 247 are from GitHub and 621 are from PyPI. Due to the volume of reports, we manually reviewed a subset of the findings, prioritizing based on the input type. We reviewed all remote and local input cases. For package-level input cases, we prioritized the popular ones—specifically, GitHub repositories with over 1,000 stars and PyPI packages with more than 10,000 monthly downloads. For the reviewed cases, we confirmed the vulnerabilities through successful exploits. Our evaluation and review show that Pyrl detects 47 exploitable zero-day class pollution vulnerabilities, highlighting both their prevalence and the lack of awareness in the Python ecosystem.

In Table 6 we show a selective list of exploitable zero-day vulnerabilities detected by Pyrl in popular Python programs. For example, Azure CLI is the official cloud management tool for Azure, with over 91.6K weekly downloads. Pyrl detected a class pollution vulnerability in it with *Agnostic-Get*$\times$*Dual-Set* primitives that can lead to

TABLE 7: [RQ2] Breakdown of Pyrl's vulnerability reports based on the input type and the pollution primitive.

	# Reported	# Checked	TP	FP
Total	868	84	47	37
Input type breakdown				
Remote Input	15	15	11	4
Local Input	23	23	15	8
Package-level Input	830	46	21	25
Pollution Primitive breakdown				
Agnostic-Get×Dual-Set	106	20	7	13
Constrained-Get×Dual-Set	17	2	1	1
Agnostic-Get×Attr-Set	27	1	0	1
Constrained-Get×Attr-Set	617	56	39	17
Agnostic-Get×Item-Set	80	0	0	0
Constrained-Get×Item-Set	21	5	0	5

token leakage and OS command injection. We reported the vulnerability to Microsoft, who patched it and assigned a CVE identifier immediately. The results also include high-profile web application frameworks such as Mesop (6.3K stars) and Taipy (18.1K stars), which are widely used for building next-generation LLM applications. Pyrl detected class pollution vulnerabilities in both frameworks, with the *Constrained-Get* primitive. These vulnerabilities can lead to DoS and other severe consequences. Both vendors, Google and Taipy Enterprise, patched the issues promptly and assigned CVE identifiers based on our reports. Pyrl also identified class pollution vulnerabilities in MLOps (Machine Learning Operations) applications, during model format transitions, merges, and patches, which are increasingly used due to techniques like LoRA. These operations introduce class pollution vulnerabilities which can be exploited using an attacker-controlled model. With class pollution, even formats that are widely believed to be safe, such as `safetensor`, can still lead to unexpected behaviors, such as a persistent Denial-of-Service (DoS) against the entire platform. We confirmed this issue in real-world cases, such as ComfyUI and sd-webui-controlnet.

We further break down all the findings by input type and pollution primitives, as shown in Table 7. In terms of input types, the number of vulnerabilities triggered through remote or local inputs is relatively small compared to those triggered through package-level inputs. This is expected, given that the Python ecosystem contains far more third-party packages than standalone applications. In terms of pollution primitives, it shows that our newly discovered type, *Constrained-Get×Attr-Set*, stands out as the most prevalent case and, based on our findings, can lead to severe consequences.

6.2. RQ2: False Positives and False Negatives

In this subsection, we evaluate the false positives and false negatives of Pyrl.

False Positives. We observed the false positives primarily due to the following reasons:

- Implicit Sanitization. While Pyrl detects explicit sanitization checks such as key matching and type validation, we also observe implicit sanitization mechanisms that prevent exploitation. These include attribute access and method calls on the target object prior to the "set" primitive. If these operations fail, an exception is raised, therefore, the "set" primitive is never executed.
- Non-feasible Paths. Pyrl handles path feasibility partially through barrier node modeling. In other cases, it conservatively over-approximates control flow by considering all branches without enforcing path constraints. As a result, false positives may occur when certain conditions are unsatisfiable.
- Non-User-Facing APIs. One of the sources of false positives in package-level input cases is the reporting on non-user-facing APIs. Those APIs are intended for internal use, even though they are exposed at the package level. Pyrl identifies package-level APIs by analyzing functions imported in the `__init__.py` file, following Python's standard packaging practices [63]. While in practice, some internal APIs are exposed in this form, which can lead to false positives.

False Negatives. Evaluating false negatives typically require a ground truth dataset, but no such known dataset exists for class pollution. Moreover, there is no prior tools existing for class pollution vulnerability detection so we could not use external reports for comparison. We therefore conducted an extensive search across the NVD database, GitHub advisories and issues, technical blogs, and other public resources using keywords such as "Python" and "Class Pollution." This yielded only two real-world cases: one in Pydash and another in PyTorch Lightning. We ran Pyrl on both cases and confirmed that both were successfully detected.

TABLE 8: [RQ3] Ablation Study on operational labels.

	GitHub Repos			PyPI packages		
	TP	FP	FPR	TP	FP	FPR
Pyrl	13	8	38.09%	8	4	33.33%
Pyrl with object taint	13	37	74.00%	8	29	78.37%
Pyrl with plain taint	13	220	94.09%	8	266	97.08%

6.3. RQ3: Ablation Study

In this subsection, we answer the research question of how much the operational labels contribute to reducing false positives in detection. We compare the full version of Pyrl with two variants: (i) Plain Taint (Pyrl-PT): A baseline that uses a single taint label for all tainted values (i.e., no distinction between taint types or operations); (ii) Object Taint (Pyrl-OT): A variant that distinguishes between tainted objects and other tainted values, following prior work on detecting prototype pollution vulnerabilities [9], [13].

We ran Pyrl and the two variants on 500 of the most popular GitHub repositories and 500 of the most downloaded PyPI packages. We selected this smaller dataset to keep the evaluation time manageable. Table 8 shows that Pyrl achieves a significantly lower false positive rate compared to Pyrl-PT and Pyrl-OT. Notably, neither variant detects any positives that Pyrl misses. We draw two key observations. First, the comparison between Pyrl-OT and Pyrl-PT demonstrates that object tainting effectively captures object resolution behavior, significantly reducing false positives. Second, the comparison between Pyrl and Pyrl-OT highlights that our label-based restriction on keys further reduces false positives, particularly in cases where not all keys involved in object resolution are fully under the attacker's control.

6.4. RQ4: Case Studies

We now showcase two zero-day class pollution vulnerabilities and describe how Pyrl detects them.

Azure CLI. Azure Command-Line Interface (CLI) [64] is the official command-line tool for managing Azure resources with interactive commands or scripts. It is vulnerable to the *Agnostic-Get×Dual-Set* type class pollution. The vulnerability lies in how Azure CLI processes the `--set` argument, as shown in Listing 3. Line 1 shows the exploit and the rest shows the vulnerable code. Specifically, the attacker-controlled value flows into function `set_properties` (line 3-10) via the `expression` parameter. Inside `set_properties`, the code first splits the key and value using `_split_key_value_pair` (line 4). Then `_get_name_path` (line 5) parses the key into a final `name` (the assignment target) and a `path` (the resolution path to the target object). At line 6, it calls function `_find_property` (line 12-18), which iteratively resolves to the final object through item access (line 15) or attribute access (line 17). Finally, depending on the object's type, it performs either item assignment (line 8) or attribute assignment (line 10).

The challenge in this case is detecting the *Agnostic-Get* primitive. The `make_snake_case` function (line 23-27) used in the attribute get operation (i.e., `getattr`) acts as a barrier: it converts keys to lowercase with underscores. This prevents the attacker from polluting arbitrary keys and blocks the T_KEY label from propagating (e.g., from `s` to `s1` at line 25). As a result, `getattr` cannot directly transfer a T_Key label to a T_OBJ with G_ATTR. That said, another second-order attribute atomic get, `obj.__dict__[name]` (Table 1), is still recognized by Pyrl. Pyrl identifies this second-order get through a separate data flow analysis before vulnerability analysis. In this case, the attribute `__dict__` does not match the regular expressions on lines 26 and 27 through a static evaluation, so it is preserved through `make_snake_case`. Thus, the `getattr` on line 18 retrieves the dictionary representation of the object, and together with the get item on line 16, this forms a second-order attribute get operation, resulting in the `instance` variable carrying the G_ATTR label. Since the get item operation on line 16 can transit a G_ITEM by itself, the `instance` variable therefore carries both G_ATTR and G_ITEM, representing an *Agnostic-Get* primitive.

Taipy. Taipy [65] is a popular Python framework for building web applications, with 18.1K stars on GitHub.

```
1  /* Exploit: az resource update --ids "X" --set "
     ↪ __class__.__init__.__globals__.sys.modules.
     ↪ subprocess.os.environ._data.COMSPEC=cmd /c calc
     ↪ " */
2
3  def set_properties(instance, expression, force_string):
4    key, value = _split_key_value_pair(expression)
5    name, path = _get_name_path(key)
6    instance = _find_property(instance, path)
7    if isinstance(instance, dict):
8      instance[name] = value
9    else:
10     setattr(instance, make_snake_case(name), value)
11
12 def _find_property(instance, path):
13   for part in path:
14     if isinstance(instance, dict):
15       instance = instance[part]
16     if hasattr(instance, make_snake_case(part)):
17       instance = getattr(instance, make_snake_case(part
             ↪ ), None)
18   return instance
19
20 snake_regex_1 = re.compile('(.)([A-Z][a-z]+)')
21 snake_regex_2 = re.compile('([a-z0-9])([A-Z])')
22
23 def make_snake_case(s):
24   if isinstance(s, str):
25     s1 = re.sub(snake_regex_1, r'\1_\2', s)
26     return re.sub(snake_regex_2, r'\1_\2', s1).lower()
27   return s
```

Listing 3: A zero-day class pollution vulnerability found in `Azure CLI` [64]. The code is simplified for explanation.

```
1  /* Exploit: WebSocket Message: 42["message",{"type":"U
     ↪ ","name":"_TpN_tpec_TpExPr_value_TPMDL_2.
     ↪ __class__.__base__.set","payload":{"value
     ↪ ":71},"client_id":"X","ack_id":"Y","
     ↪ module_context":"__main__"},null] */
2
3  def _attrsetter(obj: object, attr_str: str, value:
     ↪ object) -> None:
4    var_name_split = attr_str.split(sep=".")
5    for i in range(len(var_name_split) - 1):
6      sub_name = var_name_split[i]
7      obj = getattr(obj, sub_name)
8    setattr(obj, var_name_split[-1], value)
```

Listing 4: A zero-day class pollution vulnerability found in `Taipy`. The code is simplified for explanation.

It is vulnerable to a *Constrained-Get×Attr-Set* type class pollution. The vulnerability occurs when the server handles a client update request, shown in Listing 4, with the exploit code (line 1) and the vulnerable code (line 3-8). The handler passes an attacker-controlled dotted path (the `name` field at line 1) and a pollution value (the `value` field) to `_attrsetter` (line 3). `_attrsetter` splits the path on `"."` (line 4), resolves each segment with `getattr` in a loop, and finally calls `setattr` on the last segment (line 7).

The challenge of identifying class pollution in this case is tracking the label from T_ENUM to T_KEY. In the beginning, the parameters `attr_str` and `value` both carry T_INPUT. After the `split` at line 4, `var_name_split` is labeled T_ENUM. Next, instead of iterating the list with T_ENUM directly, the code first retrieves the loop index and then uses that index in a get-item operation to fetch the key. Pyrl handles this by tracking get item operations on an

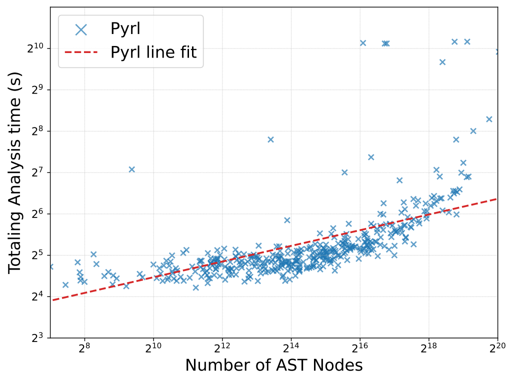


Figure 6: Total running time vs Number of AST nodes for 500 random applications.

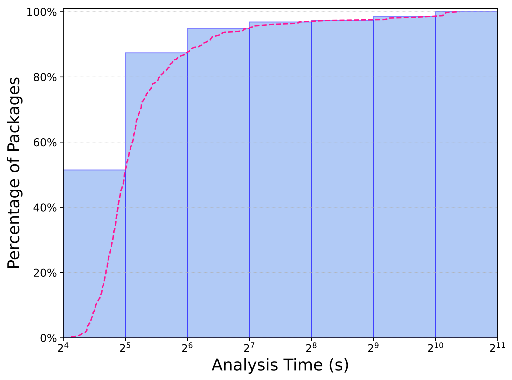


Figure 7: CDF of analysis time for 500 random applications.

object labeled `T_ENUM` where the index is the loop variable of integer type. The subsequent `getattr` on line 7 yields a `T_OBJ` that flows into `setattr` on line 7, resulting in a *Constrained-Get*×*Attr-Set* case.

6.5. RQ5: Scalability and Performance

In this subsection, we evaluate the scalability and performance overhead of Pyrl. We ran Pyrl on 500 randomly selected GitHub repositories with over 1,000 stars and recorded their analysis time. Figure 6 plots analysis time against the number of AST nodes per repository. As the number of AST nodes increases, the total analysis time grows linearly, as indicated by the fitted trend line, demonstrating that Pyrl scales to large codebases. Figure 7 further shows the cumulative distribution of analysis time across all packages.

7. Discussion

Ethics. We ensure that our study adheres to standard ethical guidelines for security research. All analyzed code was collected from publicly available sources, including open-source GitHub repositories and official PyPI packages, in compliance with the platforms' terms of service, respectively. No private or user-sensitive data was accessed.

For vulnerability disclosure, we responsibly reported all identified vulnerabilities to affected vendors and allowed a 45-day remediation period before any public disclosure. As of this writing, four vulnerabilities have been fixed and six CVE identifiers have been assigned.

Gadget Detection. Similar to prior work on prototype pollution gadgets in JavaScript [9], [10], [12]–[14], [66], serialization gadget chains in PHP [67]–[69] and Java [70]–[75], and Return-oriented Programming (ROP) / Jump-oriented Programming (JOP) gadgets in binary exploitation [76]–[78] , class pollution requires not only detecting the vulnerability but also identifying gadget chains for successful exploitation. While this paper focuses on vulnerability detection, our work identifies pollution targets and consequences, providing a foundation for future detections of class pollution gadgets.

Defense. The defense against Python class pollution can be applied along the object resolution path which consists of "get" primitives or right before the object assignment which is a "set" primitive. The first strategy is to sanitize the reflective lookup keys by filtering out the attacker-controlled ones. Through the study of real-world Python programs, we observed three types of key sanitization: (i) filtering out dangerous attributes such as `__globals__` (as seen in Pydash), (ii) checking for leading or trailing underscores (as seen in Mesop and Azure CLI), and (iii) disallowing dotted key paths (adopted by Taipy). Among these, filtering specific dangerous keys is often incomplete, as an attacker can bypass the check by selecting alternate gadgets or modifying class methods. In contrast, the latter two approaches effectively block the access of built-in attributes and prevent from reaching the unexpected object through the type hierarchy.

The second strategy of defense is to validate object types before performing assignments. This final check prevents pollution even when attacker-controlled keys resolve to the sink. For example, some applications restrict user-modifiable objects to be a fixed type, such as a dataclass, ensuring that unexpected assignments are rejected.

Class Pollution vs. Prototype Pollution. Although the name of Python class pollution is inspired by JavaScript prototype pollution and both relate to object inheritance, the internal attack working mechanisms are different. This is due to the fundamental differences in the two programming languages, e.g., different object models (prototype-based vs. class-based), object structures, attribute resolutions, reflection mechanisms, and namespace design. Specifically, JavaScript objects maintain a single namespace for all properties, accessible uniformly. By contrast, Python objects may hold separate namespaces and requires different object resolution paths. For example, a Python dictionary distinguishes between its attributes (accessed via dot notation) and its keys (accessed via brackets), requiring different operations for each namespace. Consequently, the pollution resolution path differs. In JavaScript, access is handled uniformly via the prototype chain, while in Python, attribute and item access follow different rules and orders.

These language design differences also result in distinct pollution targets. JavaScript attacks commonly aim at the root prototype so that reads of *undefined* properties inherit polluted values. In Python, pollution targets are more diverse. Any reachable runtime object (e.g., modules, classes, instances, and containers) can be polluted, and polluted values are retrieved via Python's variable, attribute, or item resolution, not only through undefined property lookups. This not only creates new exploit scenarios and broadens the potential impact (see Section 2.2.2) but also makes Python class pollution a problem distinct from JavaScript prototype pollution.

Portability of Operational Taint Analysis. The operational taint analysis proposed in this work is applicable to other dynamic, object-oriented languages by modeling data flows across different get and set operations and their relationships. Pyrl is a prototype for Python. Porting to another language requires language-specific models of (i) the data/object model (e.g., prototype vs. class, attribute vs. item namespaces, method-resolution order), (ii) reflection and dynamic lookup mechanisms, and (iii) library behaviors such as key splitting, name parsing, and enumeration/iteration. These models define the taint-propagation semantics and the core analysis applies to the target language without change.

8. Related Work

Pollution-Based Vulnerabilities. Prototype and class pollution have been examined from multiple angles. Early public attention to JavaScript prototype pollution in Node.js ecosystem was documented by Arteau [79], followed by work that developed server-side prototype pollution detection mechanisms: Kim et al. [80] built an automated detector named DAPP based on static analysis, Li et al. [11] proposed object lookup analysis that proves to be more accurate in detecting prototype pollution in Node.js, and Li et al. [49] proposed object dependence graphs to mine related vulnerabilities. Beyond detecting the vulnerability itself, researchers have also proposed methods on exploiting server-side prototype pollution and detecting their gadgets. These include Shcherbakov et al. [13] on detecting universal remote code execution (RCE) gadgets in node.js runtime, Liu et al. [12] on detecting and chaining gadgets in template engines, Shcherbakov et al. [14] on detecting gadgets at runtime, and Cornelissen et al. [66] that identified universal gadget patterns in other JavaScript runtime, i.e., Deno, further extend the attack scope beyond Node.js. On the client side, Kang et al. [9] conducted a large-scale measurement on client-side prototype pollution and later Kang et al. [10] detected client-side prototype pollution gadgets from one million real-world websites.

Recently, similar design weaknesses in other languages, particularly class pollution, have been observed by Miján [81], who provided a deep dive into Ruby class pollution caused by recursive merges, and Ouyang [16] and Qingyun [17], who reported Python class pollution attack vectors and countermeasures. Despite previous discoveries of class pollution in Python, its prevalence and severity in real-world repositories remain largely unexplored. Our work bridges this research gap by conducting the first large-scale systematic study of Python class pollution, uncovering 47 zero-day vulnerabilities.

Other Vulnerabilities in Python. Python is known to be susceptible to vulnerabilities other than class pollution, including several highlighted in the OWASP Top 10 [82], such as Cross-Site Scripting (XSS) [83] and Insecure Deserialization using modules like pickle [84] or yaml.load [85]. In addition, Supply Chain Attacks targeting PyPI [53], [86]–[90], Trojan Source Attacks [91] and vulnerabilities in CPython virtual machines [92] are also discovered and discussed. Bogaerts et al. [83] further composed a taxonomy for prevalent and well-studied Python vulnerabilities. However, none of the prior work studied class pollution systematically. In contrast, we conduct the first systematic study of Python class pollution by introducing Pyrl, the first automated tool for detecting such vulnerabilities and establishing a comprehensive taxonomy of this attack.

Python Static Analysis. Python static analysis has seen significant contributions from both academic research and industry. On the academic side, early tools like PyT [7] and Pythia [93] focus on detecting security vulnerabilities in web applications through taint analysis, with Pythia specifically targeting framework-specific issues in Django. On the industry side, production-ready tools have emerged, including Bandit [94] for common security issue checking based on AST, Meta's Pysa [6] for configurable inter-procedural taint analysis, and GitHub's CodeQL [8] for query-based security analysis. Our work proposes operational taint analysis, which propagates fine-grained, expressive labels on the target program with different operations (e.g., transforming, deriving, and merging) to track "get" and "set" primitives, and conducts the first large-scale systematic study of class pollution.

9. Conclusion

In this paper, we design and implement the first detection framework, called Pyrl, for Python class pollution via *operational taint analysis*. Pyrl tracks both "get" and "set" primitives of class pollution using operational taints, i.e., special labels that can be initiated, transformed, propagated, and merged. We ran Pyrl against over half a million real-world Python programs and libraries, resulting in 47 zero-day class pollution vulnerabilities. Examples include critical vulnerabilities in Microsoft's and Google's software. Our findings shed light on Python developers in developing secure code and in detecting and patching existing class pollution vulnerabilities.

TABLE 9: Class pollution exploits for the motivating example.

Conseq.	**Pollution Key Path**	**Example Value**	**Description**
DoS	`__class__.__getattribute__`	`1337`	Overwriting the attribute access handler to a non-callable primitive.
XSS	`__init__.__globals__.sys.modules.bs4.dammit .EntitySubstitution.CHARACTER_TO_XML_ENTITY.<`	`<script>alert(1337) </script>`	Overwriting character escape map to replace <with XSS payloads.
Authentication Bypass	`__init__.__globals__.sys.modules.django .template.backends.django.settings.SECRET_KEY`	`"13371337"`	Overwriting Django's SECRET_KEY to an attacker-controlled string.
RCE	`__init__.__globals__.sys.modules.os.environ`[1]	`{"BROWSER": "/bin/sh -c 'touch /tmp/1337'"}`	Overwriting `BROWSER` environment variable with a shell command.
	`__init__.__globals__.location_cache ._Cache__data.todo`[2]	`["antigravity", "any"]`	Modifying the module cache to load `antigravity` standard library during dynamic imports.

[1,2]: Both pollutions are needed to achieve RCE.

Acknowledgments

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361 and a Defense Advanced Research Projects Agency (DARPA) Young Faculty Award (YFA) under Grant Agreement D22AP00137-00 as well as an Amazon Research Award (ARA) 2021 and gifts from Visa Research. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, DARPA, Amazon, or Visa Research.

References

[1] "Python package index (pypi)," https://pypi.org/, accessed: 2025-05-01.

[2] GitHub, "Octoverse 2024: Ai leads python to top language as the number of global developers surges," 2024, accessed: 2025-04-24. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-2024/

[3] Abdulraheem Khaled, "Prototype pollution in python," 2023, blog. [Online]. Available: https://blog.abdulrah33m.com/prototype-pollution-in-python/

[4] D. Gilland, "pydash," https://github.com/dgilland/pydash, 2013.

[5] chilaxan, "Rce via property/class pollution due to state change endpoint in lightning-ai/pytorch-lightning," Available from huntr.com, CVE-ID CVE-2024-5452., 2024. [Online]. Available: https://huntr.com/bounties/486add92-275e-4a7b-92f9-42d84bc759da

[6] Meta, "Pysa overview — pyre," 2025, blog. [Online]. Available: https://pyre-check.org/docs/pysa-basics/

[7] S. Micheelsen and B. Thalmann, "A static analysis tool for detecting security vulnerabilities in python web applications," *Aalborg University, Aalborg University, 31st May*, 2016.

[8] O. De Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, "Keynote address:. ql for source code analysis," in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE, 2007, pp. 3–16.

[9] Z. Kang, S. Li, and Y. Cao, "Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites." in *NDSS*, 2022.

[10] Z. Kang, M. Lyu, Z. Liu, J. Yu, R. Fan, S. Li, and Y. Cao, "Follow my flow: Unveiling client-side prototype pollution gadgets from one million real-world websites," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 16–16.

[11] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting node. js prototype pollution vulnerabilities via object lookup analysis," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 268–279.

[12] Z. Liu, K. An, and Y. Cao, "Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node. js template engines for malicious consequences," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4015–4033.

[13] M. Shcherbakov, M. Balliu, and C.-A. Staicu, "Silent spring: Prototype pollution leads to remote code execution in node. js," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5521–5538.

[14] M. Shcherbakov, P. Moosbrugger, and M. Balliu, "Unveiling the invisible: Detection and evaluation of prototype pollution gadgets with dynamic taint analysis," in *Proceedings of the ACM Web Conference 2024*, 2024, pp. 1800–1811.

[15] F. Xiao, J. Huang, Y. Xiong, G. Yang, H. Hu, G. Gu, and W. Lee, "Abusing hidden properties to attack the node. js ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2951–2968.

[16] Z. Ouyang, "Research and explore of prototype pollution attack in python," in *2023 3rd Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*. IEEE, 2023, pp. 415–419.

[17] Z. Qingyun, "Exploitation and prevention of python prototype chain pollution," *Applied and Computational Engineering*, vol. 43, pp. 229–236, 2024.

[18] GitHub, Inc., "Github," https://github.com/, accessed: 2025-05-31.

[19] Python Software Foundation, "Built-in functions (getattr) — python 3.13.3 documentation," https://docs.python.org/3/library/functions.html#getattr, 2025, accessed: 2025-05-17.

[20] ——, "Data model (object.__dict__) — python 3.13.3 documentation," https://docs.python.org/3/reference/datamodel.html#object.__dict__, 2025, accessed: 2025-05-17.

[21] ——, "Data model (object.__getattribute__) — python 3.13.3 documentation," https://docs.python.org/3/reference/datamodel.html#object.__getattribute__, 2025, accessed: 2025-05-17.

[22] ——, "Built-in functions (vars) — python 3.13.3 documentation," https://docs.python.org/3/library/functions.html#vars, 2025, accessed: 2025-05-17.

[23] ——, "inspect.getmembers — python 3.13.3 documentation," https://docs.python.org/3/library/inspect.html#inspect.getmembers, 2025, accessed: 2025-05-17.

[24] ——, "operator.attrgetter — python 3.13.3 documentation," https://docs.python.org/3/library/operator.html#operator.attrgetter, 2025, accessed: 2025-05-17.

[25] ——, "inspect.getattr_static — python 3.13.3 documentation," https://docs.python.org/3/library/inspect.html#inspect.getattr_static, 2025, accessed: 2025-05-17.

[26] ——, "Built-in functions (dir) — python 3.13.3 documentation," https://docs.python.org/3/library/functions.html#dir, 2025, accessed: 2025-05-17.

[27] ——, "inspect.getmembers_static — python 3.13.3 documentation," https://docs.python.org/3/library/inspect.html#inspect.getmembers_static, 2025, accessed: 2025-05-17.

[28] ——, "Mapping types — dict — python 3.13.3 documentation," https://docs.python.org/3/library/stdtypes.html#mapping-types-dict, 2025, accessed: 2025-05-17.

[29] ——, "Sequence types — list, tuple, range — python 3.13.3 documentation," https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range, 2025, accessed: 2025-05-17.

[30] ——, "dict.get() — python 3.13.3 documentation," https://docs.python.org/3/library/stdtypes.html#dict.get, 2025, accessed: 2025-05-17.

[31] ——, "Mutable sequence types — python 3.13.3 documentation," https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types, 2025, accessed: 2025-05-17.

[32] ——, "dict.setdefault() — python 3.13.3 documentation," https://docs.python.org/3/library/stdtypes.html#dict.setdefault, 2025, accessed: 2025-05-17.

[33] ——, "Data model (object.__getitem__) — python 3.13.3 documentation," https://docs.python.org/3/reference/datamodel.html#object.__getitem__, 2025, accessed: 2025-05-17.

[34] ——, "operator.getitem — python 3.13.3 documentation," https://docs.python.org/3/library/operator.html#operator.getitem, 2025, accessed: 2025-05-17.

[35] ——, "operator.itemgetter — python 3.13.3 documentation," https://docs.python.org/3/library/operator.html#operator.itemgetter, 2025, accessed: 2025-05-17.

[36] ——, "operator.__getitem__ — python 3.13.3 documentation," https://docs.python.org/3/library/operator.html#operator.__getitem__, 2025, accessed: 2025-05-17.

[37] ——, "Built-in functions (eval) — python 3.13.3 documentation," https://docs.python.org/3/library/functions.html#eval, 2025, accessed: 2025-05-17.

[38] ——, "Built-in functions (exec) — python 3.13.3 documentation," https://docs.python.org/3/library/functions.html#exec, 2025, accessed: 2025-05-17.

[39] ——, "Built-in functions (setattr) — python 3.13.3 documentation," https://docs.python.org/3/library/functions.html#setattr, 2025, accessed: 2025-05-17.

[40] ——, "Data model (object.__setattr__) — python 3.13.3 documentation," https://docs.python.org/3/reference/datamodel.html#object.__setattr__, 2025, accessed: 2025-05-17.

[41] ——, "dict.update() — python 3.13.3 documentation," https://docs.python.org/3/library/stdtypes.html#dict.update, 2025, accessed: 2025-05-17.

[42] ——, "Data model (object.__setitem__) — python 3.13.3 documentation," https://docs.python.org/3/reference/datamodel.html#object.__setitem__, 2025, accessed: 2025-05-17.

[43] ——, "operator.setitem — python 3.13.3 documentation," https://docs.python.org/3/library/operator.html#operator.setitem, 2025, accessed: 2025-05-17.

[44] ——, "operator.__setitem__ — python 3.13.3 documentation," https://docs.python.org/3/library/operator.html#operator.__setitem__, 2025, accessed: 2025-05-17.

[45] ——, "The python standard library," https://docs.python.org/3/library/, 2024, accessed: 2025-05-05.

[46] ——, "Pep 3102 – keyword-only arguments," https://peps.python.org/pep-3102/, 2025, accessed: 2025-09-30.

[47] ——, "Data model — python 3.13.3 documentation," https://docs.python.org/3/reference/datamodel.html#data-model, 2025, accessed: 2025-05-17.

[48] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: feedback-driven static analysis of node. js applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 455–465.

[49] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node. js vulnerabilities via object dependence graph and query," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 143–160.

[50] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1059–1076.

[51] D. Cassel, N. Sabino, M.-C. Hsu, R. Martins, and L. Jia, "Nodemedic-fine: Automatic detection and exploit synthesis for node. js vulnerabilities," in *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS'25). doi*, vol. 10, 2025.

[52] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1046–1058.

[53] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023.

[54] R. Wang, S. Xu, X. Ji, Y. Tian, L. Gong, and K. Wang, "An extensive study of the effects of different deep learning models on code vulnerability detection in python code," *Automated Software Engineering*, vol. 31, no. 1, p. 15, 2024.

[55] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.

[56] GitHub, "Arbitrary code execution via crafted keras config for model loading," Available from github.com, CVE-ID CVE-2025-1550., 2025. [Online]. Available: https://github.com/advisories/GHSA-48g7-3x6r-xfhp

[57] ——, "Pydash command injection vulnerability," Available from github.com, CVE-ID CVE-2023-26145., 2023. [Online]. Available: https://github.com/advisories/GHSA-8mjr-6c96-39w8

[58] ——, "Paddlepaddle vulnerable to remote code execution," Available from github.com, CVE-ID CVE-2024-0917., 2024. [Online]. Available: https://github.com/advisories/GHSA-mrmm-qmrj-xgp6

[59] ——, "Pydantic regular expression denial of service," Available from github.com, CVE-ID CVE-2024-3772., 2024. [Online]. Available: https://github.com/advisories/GHSA-mr82-8j83-vxmv

[60] Adam Hill, "Unicorn: A magical full-stack framework for django," 2023, website. [Online]. Available: https://www.django-unicorn.com/

[61] Leonard Richardson, "Beautiful soup: We called him tortoise because he taught us," 2025, website. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/

[62] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*. Springer, 1999, pp. 193–207.

[63] PyPA, “Packaging python projects — python packaging user guide,” https://packaging.python.org/en/latest/tutorials/packaging-projects/, 2025, accessed: 2025-05-17.

[64] “Azure command-line interface (cli) documentation,” https://learn.microsoft.com/en-us/cli/azure/?view=azure-cli-latest, Microsoft, 2025, accessed: YYYY-MM-DD.

[65] “Taipy: Python framework for building data apps,” https://taipy.io/, Taipy, 2025, accessed: 2025-09-30.

[66] E. Cornelissen, M. Shcherbakov, and M. Balliu, “{GHunter}: Universal prototype pollution gadgets in {JavaScript} runtimes,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3693–3710.

[67] B. Pang, Y. Zhang, M. Gao, J. Zhang, L. Chen, M. Zhang, and G. Liang, “Pfortifier: Mitigating php object injection through automatic patch generation,” in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2025, pp. 918–933.

[68] Y. David, N. Christou, A. D. Kellas, V. P. Kemerlis, and J. Yang, “Quack: Hindering deserialization attacks via static duck typing,” in *the Network and Distributed System Security Symposium (NDSS)*, 2024.

[69] S. Park, D. Kim, S. Jana, and S. Son, “{FUGIO}: Automatic exploit generation for {PHP} object injection vulnerabilities,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 197–214.

[70] X. Chen, B. Wang, Z. Jin, Y. Feng, X. Li, X. Feng, and Q. Liu, “Tabby: Automated gadget chain detection for java deserialization vulnerabilities,” in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2023, pp. 179–192.

[71] B. Chen, L. Zhang, X. Huang, Y. Cao, K. Lian, Y. Zhang, and M. Yang, “Efficient detection of java deserialization gadget chains via bottom-up gadget search and dataflow-aided payload construction,” in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 3961–3978.

[72] B. Kreyssig, T. Riom, S. Houy, A. Bartel, and P. McDaniel, “Deserialization gadget chains are not a pathological problem in android: an in-depth study of java gadget chains in aosp,” *arXiv preprint arXiv:2502.08447*, 2025.

[73] S. Cao, X. Sun, X. Wu, L. Bo, B. Li, R. Wu, W. Liu, B. He, Y. Ouyang, and J. Li, “Improving java deserialization gadget chain mining via overriding-guided object generation,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 397–409.

[74] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma *et al.*, “Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing,” in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2726–2743.

[75] P. Srivastava, F. Toffalini, K. Vorobyov, F. Gauthier, A. Bianchi, and M. Payer, “Crystallizer: A hybrid path analysis framework to aid in uncovering deserialization vulnerabilities,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1586–1597.

[76] N. Christou, A. J. Gaidis, V. Atlidakis, and V. P. Kemerlis, “Eclipse: Preventing speculative memory-error abuse with artificial data dependencies,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3913–3927.

[77] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code-reuse attack,” in *Proceedings of the 6th ACM symposium on information, computer and communications security*, 2011, pp. 30–40.

[78] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012.

[79] O. Arteau, “Prototype pollution attack in nodejs application,” https://github.com/HoLyVieR/prototype-pollution-nsec18/blob/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf, 2018, online; Accessed on 18 Feb 2021.

[80] H. Y. Kim, J. H. Kim, H. K. Oh, B. J. Lee, S. W. Mun, J. H. Shin, and K. Kim, “Dapp: automatic detection and analysis of prototype pollution vulnerability in node. js modules,” *International Journal of Information Security*, vol. 21, no. 1, pp. 1–23, 2022.

[81] R. Miján, “Class pollution in ruby: A deep dive into exploiting recursive merges,” https://blog.doyensec.com/2024/10/02/class-pollution-ruby.html, October 2024, accessed: 2025-05-27.

[82] OWASP Foundation, “OWASP Top 10:2021 – The Ten Most Critical Web Application Security Risks,” https://owasp.org/Top10/, 2021, accessed: 2025-06-05.

[83] F. C. Bogaerts, N. Ivaki, and J. Fonseca, “A taxonomy for python vulnerabilities,” *IEEE Open Journal of the Computer Society*, vol. 5, no. 01, pp. 368–379, 2024.

[84] P. S. Foundation, “pickle — python object serialization,” https://docs.python.org/3/library/pickle.html, 2025, accessed: 2025-06-05.

[85] OWASP Foundation, “Deserialization cheat sheet,” https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html, 2021, accessed: 2025-06-05.

[86] P. Ladisa, H. Plate, M. Martinez, and O. Barais, “Sok: Taxonomy of attacks on open-source software supply chains,” in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1509–1526.

[87] T. Zheng, H. Lan, and B. Li, “Be careful with pypi packages: You may unconsciously spread backdoor model weights,” *Proceedings of Machine learning and systems*, vol. 5, pp. 145–163, 2023.

[88] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, “Towards measuring supply chain attacks on package managers for interpreted languages,” *arXiv preprint arXiv:2002.01139*, 2020.

[89] A. Bagmar, J. Wedgwood, D. Levin, and J. Purtilo, “I know what you imported last summer: A study of security threats in thepython ecosystem,” *arXiv preprint arXiv:2102.06301*, 2021.

[90] D. Gonzalez, T. Zimmermann, P. Godefroid, and M. Schäfer, “Anomalicious: Automated detection of anomalous and potentially malicious commits on github,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 258–267.

[91] N. Boucher and R. Anderson, “Trojan source: Invisible vulnerabilities,” in *32nd USENIX security symposium (USENIX Security 23)*, 2023, pp. 6507–6524.

[92] C. Jiang, B. Hua, W. Ouyang, Q. Fan, and Z. Pan, “Pyguard: Finding and understanding vulnerabilities in python virtual machines,” in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 468–475.

[93] L. Giannopoulos, E. Degkleri, P. Tsanakas, and D. Mitropoulos, “Pythia: identifying dangerous data-flows in django-based applications,” in *Proceedings of the 12th European Workshop on Systems Security*, 2019, pp. 1–6.

[94] Bandit Developers, “Bandit,” https://bandit.readthedocs.io/en/latest/, accessed: 2025-09-30.

Appendix A. Exploits for Motivating Example

Table 9 shows the payloads used in the exploits against the motivating example.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper presents Pyrl, a static analysis tool that uses operational taint tracking to detect class pollution vulnerabilities in Python code. This tool is used to perform the first large-scale empirical study of such vulnerabilities, discovering 47 zero-day vulnerabilities across both open source and commercial software products used by major companies.

B.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Provides a New Data Set for Public Use
- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The paper presents a new definition and taxonomy for Python class pollution vulnerabilities.
2) Pyrl is shown to be scalable and effective at discovering class pollution vulnerabilities in real Python code.