CrossFit

Demystifying VM Callback Bugs in Interpreters

Chibin Zhang, Qiang Liu, Mathias Payer

EPFL hexhive

FSE 2026

Motivation

- Scripting languages promise extensive features and safety, but are implemented in **C/C++**.
- Bugs in scripting engines undo memory safety: **RCE, sandbox escape**.
- Recurring issue in **bug trackers**: script callbacks breaking interpreter assumptions.

Motivating example

```
class poc():
  def __eq__(self, other):
    l.clear()
    return NotImplemented

l = [poc(), poc()]
3 in l
```

Clearing a list while iterating through it

Motivating example

```python
class poc():
  def __eq__(self, other):
    l.clear()
    return NotImplemented

l = [poc(), poc()]
3 in l
```

Clearing a list while iterating through it

```c
list_count_impl(...) {
  for (i = start; i < stop; i++) {
    // self->ob_item[i] can uaf
    int cmp =
PyObject_RichCompareBool(
      self->ob_item[i], value, Py_EQ
    );
  }
}
```

*Native implementation of the **in** operator*

Motivating example

RichCompareBool calls __eq__

```python
class poc():
  def __eq__(self, other):
    l.clear()
    return NotImplemented

l = [poc(), poc()]
3 in l
```

Clearing a list while iterating through it

```c
list_count_impl(...) {
  for (i = start; i < stop; i++) {
    // self->ob_item[i] can uaf
    int cmp =
PyObject_RichCompareBool(
      self->ob_item[i], value, Py_EQ
    );
  }
}
```

*Native implementation of the **in** operator*

Motivating example

```python
class poc():
  def __eq__(self, other):
    l.clear()
    return NotImplemented

l = [poc(), poc()]
3 in l
```

Clearing a list while iterating through it

RichCompareBool calls __eq__

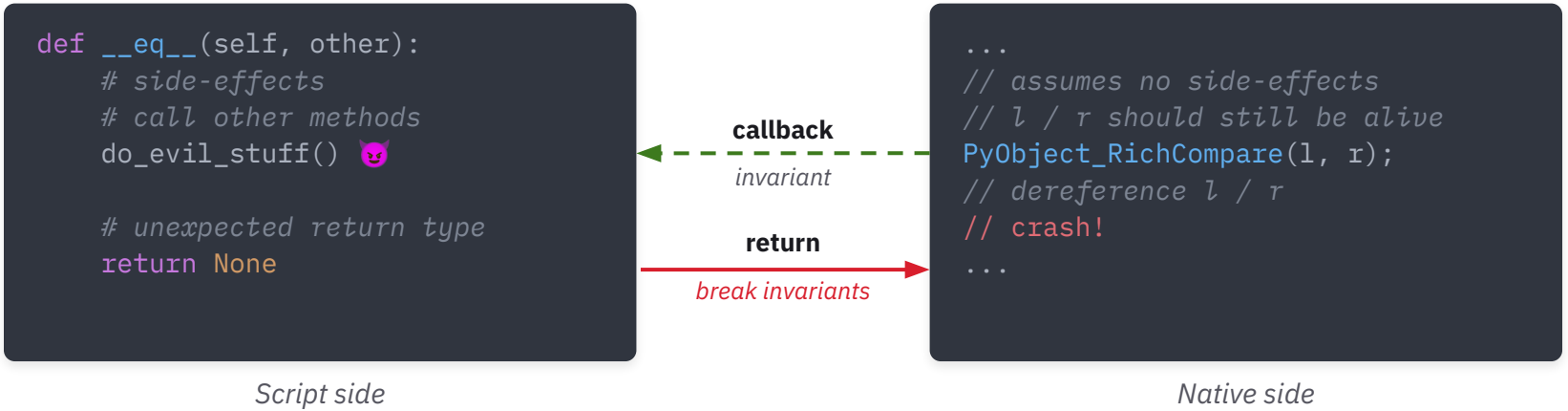
```c
list_count_impl(...) {
  for (i = start; i < stop; i++) {
    // self->ob_item[i] can uaf
    int cmp =
PyObject_RichCompareBool(
      self->ob_item[i], value, Py_EQ
    );
  }
}
```

*Native implementation of the **in** operator*

💥 **ob_item freed inside __eq__, native deref is a use-after-free**

Defining callback bugs

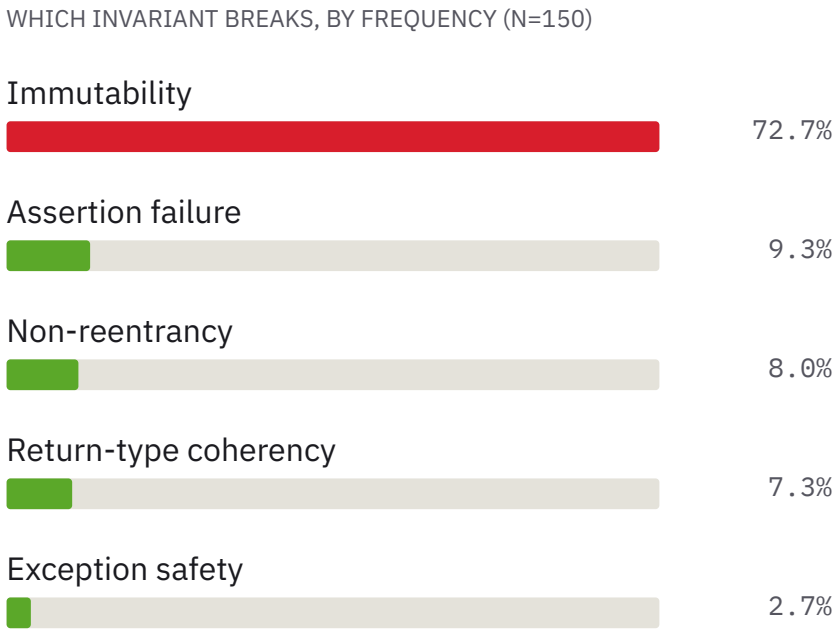
- **Callback bug**: user code breaks an interpreter *invariant* during an implicit callback.
- The callback is a **magic method**; the runtime can't defend against it.



Preliminary studies

- **Dataset**: 150 bug reports with working PoCs, 50 per language, mined from their issue trackers.
- **Immutability dominates:** most crashes free or resize state the runtime still holds.

We distill four bug triggering preconditions from this survey.



Four preconditions to find bugs

```
class poc():                    # P2
  def __eq__(self, other):
    l.clear()                   # P3
    return NotImplemented

l = [poc(), poc()]
3 in l                          # P1
```

```
// native loop over ob_item
cmp = RichCompareBool(
    self->ob_item[i], ...);  # P4
```

P1 **Reach:** reach a callback (an entrance op)

P2 **Control:** user-controlled object

P3 **Mutate:** trigger a side effect

P4 **Use:** mishandled post-callback access

Triggering a callback bug requires all four preconditions

Why existing tools miss these bugs

- **Static analysis is native-side only.** It can flag the callsite (*Reach*), but not what a user `__destruct()` will do (*Mutate*), and produces no PoC.
- **Fuzzers can't satisfy all four.** Nautilus and PolyGlot reach a callsite or define a class, but rarely trigger a user object whose magic method *also* mutates.

To trigger these bugs, a tool must reason about both the script and native sides at once.

CrossFit Design

Context link analysis

*finds **Reach** and **Use***

Maps the script operation to the magic method it can invoke, and the native callsites with post-callback access.

Targeted generation

*manufactures **Control** and **Mutate***

Builds user classes with magic methods and injects side effects, aimed at those callsites. Every reported bug is a real crash.

Static analysis finds the bridges from script to native. The fuzzer crosses them on purpose.

Context link analysis

Tracing one PHP entrance operation down to its magic method:

L0 · script

```
unset($obj)
```

↓ *map script operation to its native handler (manual, ~100 opcodes)*

L1 · bytecode

```
ZEND_FETCH_DIM_UNSET
```

↓ *automated LLVM-IR call-graph BFS*

L2 · native

```
...HANDLER() -> spl_fixedarray_object_unset_dimension() -> zval_ptr_dtor()
```

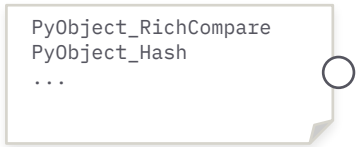
↓ *map magic method to its native invokers (manual, per language)*

L0' · script

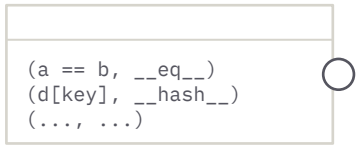
```
__destruct()
```

The native call-graph search is automated. The two ends are small, one-time manual maps per language.

Targeted generation: from callsites to fuzz inputs



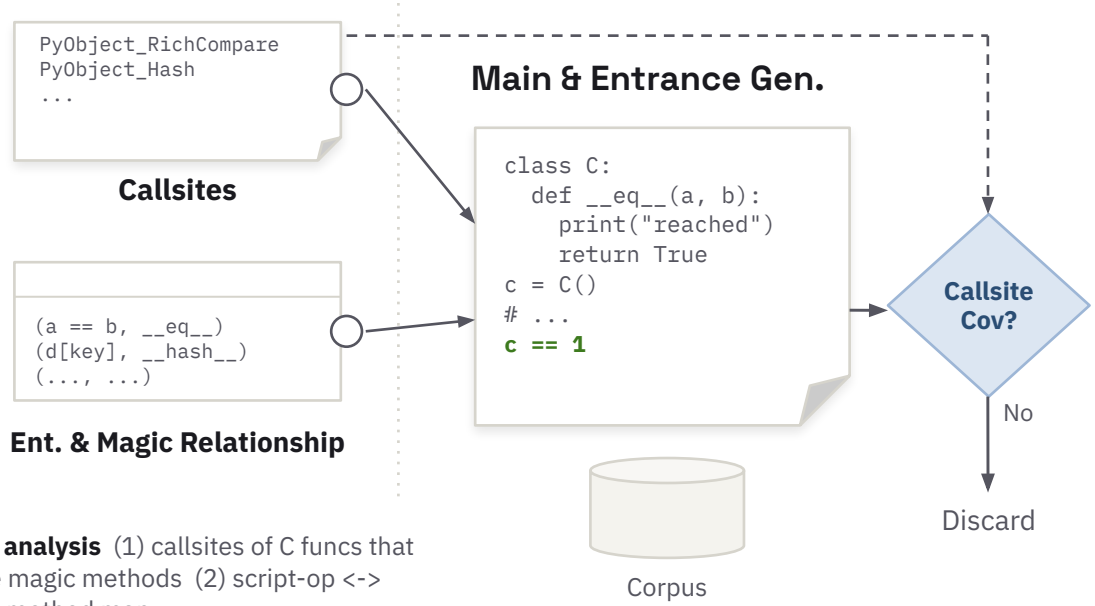
Callsites



Ent. & Magic Relationship

Static analysis (1) callsites of C funcs that invoke magic methods (2) script-op <-> magic-method map

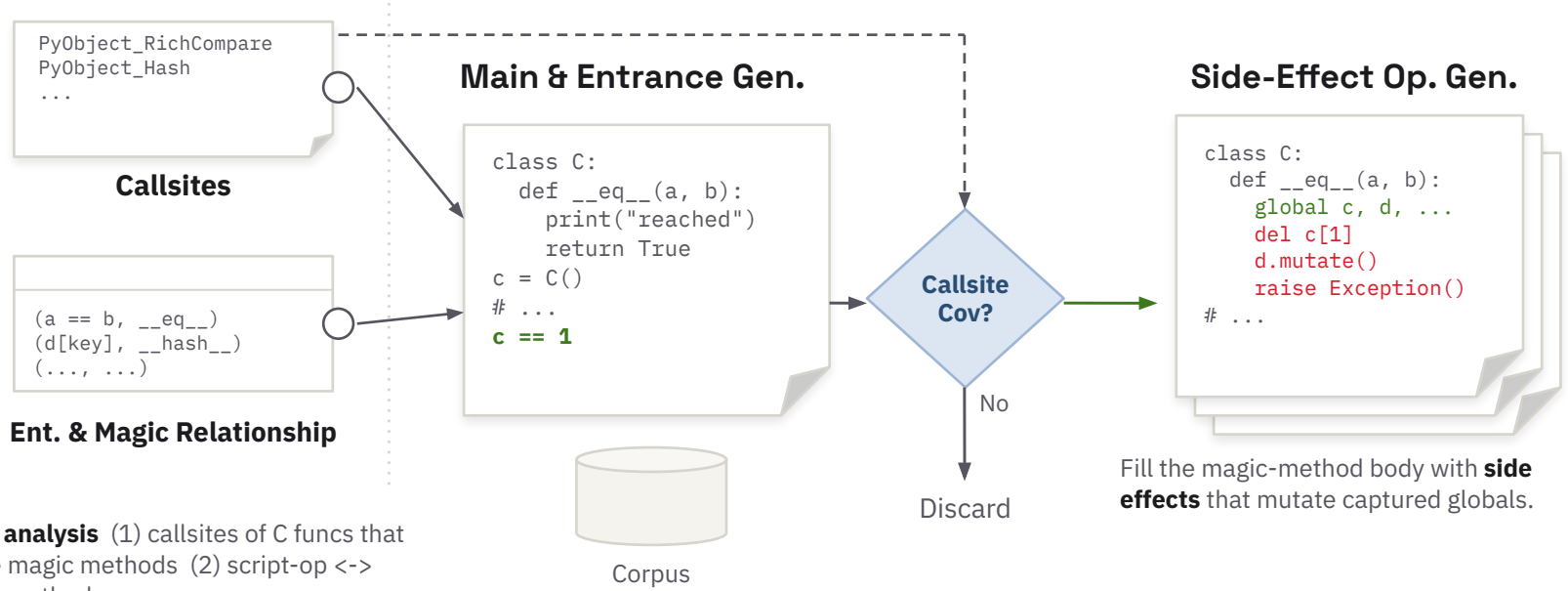
Phase 1: program skeletons



Static analysis (1) callsites of C funcs that invoke magic methods (2) script-op <-> magic-method map

Keep only skeletons that reach **new callsite coverage**; otherwise discard.

Phase 2: side effects



Static analysis (1) callsites of C funcs that invoke magic methods (2) script-op <-> magic-method map

Keep only skeletons that reach **new callsite coverage**; otherwise discard.

Fill the magic-method body with **side effects** that mutate captured globals.

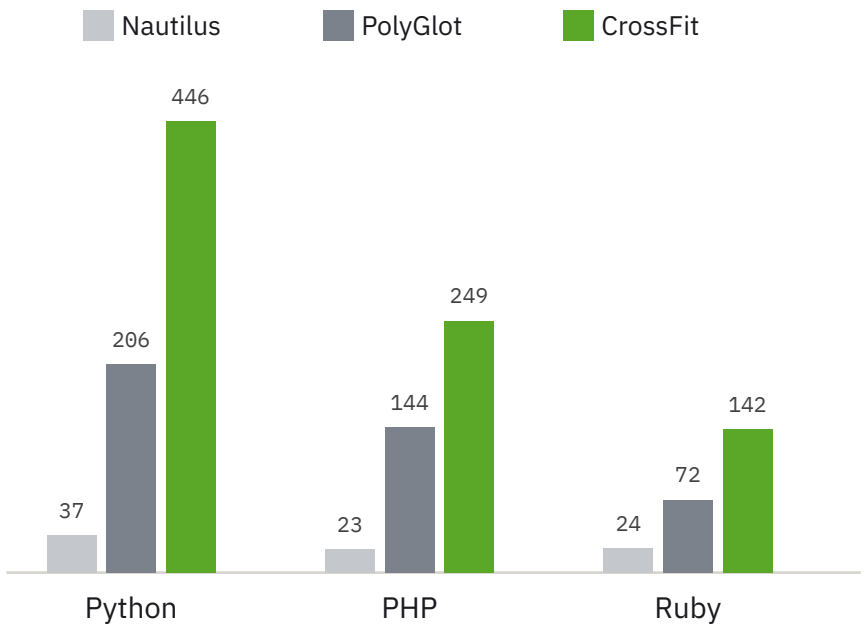
Evaluation: setup

- **Targets:** Python 3.13, PHP 8.4, Ruby 3.4, all extensions; **ASan** oracle.
- 24 h **× 8 repeats** per fuzzer-target pair.
- Baselines: PolyGlot + Nautilus (only fuzzers covering all three); added try/except guards.
- Nautilus: hand-crafted **~100 class/function grammar rules** per language (best-case boost).

Four questions: RQ1 coverage, RQ2 ablation, RQ3 new bugs, RQ4 reproduction.

Effective callsite coverage: ~2x the baselines

- **Edge coverage misleads** here: we distill the corpus, so edge drops on purpose.
- Right metric: **effective callsite coverage** (callsites where a user magic method actually fires).
- **+12.04%** callsites vs PolyGlot; **1.96x** effective coverage on average vs Nautilus.



Effective callsite coverage (count)

20 new bugs, mostly use-after-free

- **20 new bugs**: 12 PHP, 7 Ruby, 1 Python.
- Latent for years.
- All confirmed and fixed upstream.
- Collected a **150-PoC dataset**
- CrossFit reproduces **73.2% (60/82)**.

Lang	Callback	Crash site	Type
PHP	`__toString`	`zend_std_compare_objects`	UAF
PHP	`__serialize`	`ps_srlzr_encode_php`	UAF
Ruby	`hash`	`rb_ary_difference_multi`	OOB
Ruby	`block`	`st_general_foreach`	UAF
CPython	`__eq__`	`_odict_keys_equal`	UAF

5 of 20 shown; all confirmed and fixed. Full list in the artifact.

Ablation Study

- **CrossFit-NoLink** (random ops): PolyGlot-level coverage, **0 bugs**.
- **CrossFit-NoCorpus**: lower coverage, still **6/12** PHP bugs.
- PolyGlot, Nautilus, whole campaign: **0 bugs**.

Reaching ≠ triggering

`unset(new stdClass())` hits `zval_ptr_dtor()` but runs no user `__destruct()`. Coverage counts it; nothing happens.

Context link analysis plus targeted generation contribute to the difference, not the corpus.

Conclusion: CrossFit

EPFL hexhive

- **Formalized** callback bugs: four preconditions, five invariants.
- **Built** a two-tier tool: context link analysis + targeted generation.
- **Four preconditions:** Reach, Control, Mutate, Use.
- **Found** 20 new bugs (mostly UAF, all fixed) and a 150-PoC benchmark.

ARTIFACT `github.com/HexHive/crossfit-artifact`