CrossFit: Demystifying VM Callback Bugs in Interpreters

CHIBIN ZHANG, EPFL, Switzerland
QIANG LIU, EPFL, Switzerland
MATHIAS PAYER, EPFL, Switzerland

Scripting languages like Python, Ruby, or PHP are integral to modern software development. Despite security measures like memory safety and sandboxing, vulnerabilities within these engines can lead to critical issues such as remote code execution or sandbox escapes. A particularly pervasive class of vulnerabilities is *callback bugs*, which occur when user-defined callbacks violate runtime invariants, such as freeing an object still in use (can be reached through live pointers) or modifying a data structure during traversal. These violations can result in severe consequences, including use-after-free, null-pointer dereferences, and type confusion, often leading to crashes, memory corruption, or even exploitable vulnerabilities. Detecting callback bugs remains challenging due to a lack of general understanding, as they have not been formally characterized or systematically studied. As such, existing tools lack the ability to (1) establish clear links between script-side callbacks and their native-side invokers, and (2) generate scripts that systematically satisfy preconditions required to trigger these bugs.

We propose CrossFit, a novel 2-tier approach combining static analysis and targeted fuzzing to systematically discover callback bugs. CrossFit first establishes links between script-side callbacks and their native-side invokers through context link analysis, enabling targeted exploration of high-risk code paths. It then generates proof-of-concept scripts with custom classes and magic methods, introducing side-effect operations to violate runtime invariants. Our evaluation shows that CrossFit effectively outperforms existing tools by up to **12.04%** in terms of callsite coverage (i.e., potential sites where callback bugs may occur). We also identified **20** new bugs in Python, Ruby, and PHP, many of which are severe memory corruptions. Moreover, we provide a comprehensive benchmark totaling 150 proof-of-concepts to improve interpreter security.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Software Testing, Fuzzing, Interpreters, Programming Languages

ACM Reference Format:
Chibin Zhang, Qiang Liu, and Mathias Payer. 2026. **CrossFit: Demystifying VM Callback Bugs in Interpreters.** ***Proc. ACM Softw. Eng.*** **3, FSE, Article FSE104 (July 2026), 22 pages. https://doi.org/10.1145/3808111**

1 Introduction

Scripting languages like Python, Ruby, and PHP are fundamental to modern software development, powering diverse applications from web services to machine learning pipelines. Their flexibility, ease of use, and rich standard libraries enable rapid development, but they also introduce distinctive security challenges. These languages rely on sophisticated execution engines that convert script source code into bytecode, which is then interpreted or Just-In-Time (JIT) compiled into machine code. While these engines provide critical security measures such as memory safety and sandboxing, vulnerabilities within the runtime environments themselves can compromise these protections, leading to severe security issues like remote code execution or sandbox escapes [6].

Authors' Contact Information: Chibin Zhang, EPFL, Lausanne, Switzerland, chibin.zhang@epfl.ch; Qiang Liu, EPFL, Lausanne, Switzerland, cyruscyliu@gmail.com; Mathias Payer, EPFL, Lausanne, Switzerland, mathias.payer@nebelwelt.net.

ACM 2994-970X/2026/7-ARTFSE104
https://doi.org/10.1145/3808111

A particularly insidious class of vulnerabilities is *callback bugs*, which arise from complex interactions between the interpreter's native code and user-defined script code. Callback bugs occur when user-defined callbacks, such as magic methods (e.g., `__destruct()` or `__toString()`), violate runtime invariants, such as freeing an internal object still in use or modifying a data structure during its traversal. These violations can lead to severe consequences, including use-after-free (UAF), null-pointer dereferences, and type confusion, often resulting in memory corruption that may manifest as crashes or enable security exploits. For instance, `js-vuln-db` [4] documents 179 JavaScript engine vulnerabilities, with 69 (38.5%) being callback bugs that received CVEs, bug bounties, and were exploited in the wild.

Despite their prevalence and impact, callback bugs remain understudied, as they are challenging to detect. These challenges stem from two key limitations. *Lack of Clear Links Between Script and Native Code*: Triggering a callback bug requires satisfying four preconditions: (1) control flow must reach a callback in the script, (2) the callback must involve a user-controlled object, (3) the callback must introduce unintended side effects (e.g., freeing memory), and (4) the native code must mishandle post-callback access to the affected objects. Existing static analyzers [1, 11, 12] operate solely on the native side and cannot predict how user-defined script code can introduce unintended side effects, leading to runtime invariant violations. *Uncoordinated Script Generation*: Existing fuzzers [9, 13, 16] lack the ability to generate scripts that systematically satisfy all four preconditions. For example, while they may generate code that defines a `__destruct()` method, they often fail to link it to the appropriate native callsite or introduce the necessary side effects to trigger bugs, as the implementation detail is rarely publicly documented and can only be obtained through the source code.

To address these challenges, we present CrossFit, a novel approach that combines static analysis and targeted fuzzing to systematically detect callback bugs. CrossFit leverages *context link analysis* to bridge the causal gap by establishing a correspondence between script-side callbacks (e.g., magic methods) and their native-side invokers, which are specific locations in native code where callbacks are invoked (callsites). We manually identify callback invokers among stable public APIs through a frequency-based selection, and our analysis operates at the LLVM IR level to handle complex C constructs transparently. CrossFit then uses *targeted script generation* to create proof-of-concepts (PoCs) that systematically satisfy all four preconditions for callback bugs. By prioritizing the generation of custom classes with magic methods and introducing side-effect operations, CrossFit explores high-risk code paths that are often missed by existing tools, achieving superior effective callsite coverage.

We evaluate CrossFit on the latest versions of PHP, Python, and Ruby [7]. Our results demonstrate that CrossFit outperforms state-of-the-art fuzzers like Polyglot [13] and Nautilus [9] in callsite coverage, achieving improvements of **12.04%** when utilizing a corpus and **1.85x** when starting from scratch. During fuzzing, CrossFit discovered **12** new bugs in PHP, **1** new bug in Python, and **8** new bugs in Ruby, predominantly severe use-after-free memory corruptions. These bugs were missed by existing fuzzers and code audits, and all have been confirmed and fixed by developers. To validate CrossFit's ability to reproduce known bugs, we constructed a curated benchmark of 150 proof-of-concept scripts (50 each for Python, PHP, and Ruby), following ground-truth benchmarking practices like MAGMA [18]. CrossFit efficiently reproduced 73.8% of these known bugs, demonstrating its effectiveness in uncovering latent vulnerabilities. The key contributions of this work are as follows:

- We are the first to systematically characterize and formalize callback bugs, a common and prevalent bug pattern found across many scripting languages.

```php
<?php
class C {
    function __destruct() {
        global $arr;
        /* ③ Triggering unintended side effects */
        $arr->setSize(0);
    }
}
$arr = new SplFixedArray(2);

/* ② Involving user-controlled objects */
$arr[0] = new C;

/* ① Reaching a callback */
unset($arr[0]);
```

(a) Script that triggers a use-after-free in PHP.

```c
static void spl_fixedarray_object_unset_dimension(
        spl_fixedarray_object *intern,
        zval *offset
) {
    zend_long index;
    index = spl_offset_convert_to_long(offset);
    if (EG(exception)) { return; }
    if (index >= intern->array.size) /*...*/
    else {
        /* ① The callback is __destruct() */
        zval_ptr_dtor(&intern->array.elements[index]);
        /* ④ Mishandling post-callback access, UAF */
        ZVAL_NULL(&intern->array.elements[index]);
    }
}
```

(b) Native-side code of `unset_dimension`.

Fig. 1. Example of callback bug in PHP

- We propose an automated static analysis and fuzzing tool for detecting callback bugs, identifying **20** bugs in PHP, Ruby and Python.
- We provide a comprehensive ground-truth benchmark of 150 PoCs documenting this bug class. Our benchmark and implementation are open-source for further research [33].

2 Background and Motivating Example

Modern scripting languages like Python, JavaScript, PHP, and Ruby rely on *bytecode virtual machines (VMs)* to execute code across diverse platforms. The VM parses the script source code into an abstract syntax tree (AST) representation that is further compiled to a sequence of platform-agnostic bytecode, then executes the bytecode using a main evaluation loop, simulating a virtual CPU with program counters, registers, and a stack.

These VMs are famous for their extensive *standard libraries* that are generally implemented natively with user-defined *callbacks*. The standard libraries provide rich functionalities ranging from data structures like arrays and dictionaries to algorithms like sorting and searching. While some of these functionalities could be implemented in the scripting language itself, they are often written in native code (e.g., C/C++) for performance reasons. For example, sorting algorithms are typically implemented natively to avoid the overhead of dynamic dispatch in script code. However, to maintain flexibility, these native implementations often allow users to define custom behavior through callbacks. For instance, a sorting function might accept a user-defined comparison callback to determine the order of elements.

This interplay between native code and user-defined callbacks is a hallmark of scripting language interpreters, enabling both high performance and developer flexibility. However, if the user-defined callback violates the interpreter's assumptions, it can introduce subtle and severe vulnerabilities. When the interpreter invokes a user-defined callback, it temporarily transfers control from the native side to the script side. The interpreter implicitly assumes that callback will not violate its assumptions, but user-defined callbacks may, in fact, violate these assumptions. These bugs are particularly insidious because they arise from native-script interaction, making them difficult to detect with traditional testing or fuzzing tools. Existing fuzzers rely on luck to generate the corresponding magic method and entrance operation that must work in coordination, without understanding the semantic relationship between script-level operations (like `unset()`) and their native-side callback invokers. Moreover, callbacks are prevalent in standard library functions like sorting, serialization, and iteration, making these bugs both common and highly impactful, often resulting in severe vulnerabilities [5, 27].

Figure 1 showcases a use-after-free introduced by a callback in PHP. The script defines a class with a destructor callback that modifies a global array by setting its size to 0. The execution follows this sequence: ① the script calls an entrance operation (unset) that reaches the callback invoker (`zval_ptr_dtor()`), which ② operates on a user-controlled object containing a magic method. When the native interpreter invokes the callback invoker to destroy the object, this magic method (`__destruct()`) ③ triggers unintended side effects that free the array's internal memory, violating the interpreter's assumptions. Finally, ④ the native code mishandles post-callback access by attempting to access the freed element, resulting in a use-after-free vulnerability.

3 Characteristics of Callback Bugs

Section 2 provided an intuitive example of a callback bug in PHP. To obtain a *general* understanding of this bug class, this section proceeds in the following order: we first establish the key terminology used throughout the paper (Section 3.1); we then describe the methodology we used to collect and analyze a dataset of historical callback bugs (Section 3.2); based on that dataset we derive the five runtime invariants that callback bugs violate (Section 3.3) and give a formal definition of "callback bug"; we then enumerate the four preconditions that must jointly hold for a callback bug to trigger (Section 3.4); and finally we discuss why existing tools struggle with this bug class (Section 3.5).

3.1 Key Concepts

We first establish consistent terminology used throughout this paper by defining the following key concepts:

Entrance Operations. Script-side operations that can trigger callbacks, including (1) binary operations (e.g., +, ==), (2) unary operations (e.g., -, `unset()`), and (3) method/function calls and control structures (e.g., `serialize()`, `foreach`).

Callback Invokers. Native-side C functions in the interpreter that are dispatched by certain entrance operations and eventually lead to the execution of script-side magic methods. These form the bridge between script-level operations and callback execution: when an entrance operation is executed (e.g., `unset($obj)`), it triggers a chain of native function calls that culminate in a callback invoker (e.g., `zval_ptr_dtor()`) invoking the corresponding magic method (e.g., `__destruct()`).

Callback Invoker Sites (Callsites). Specific locations in the native code where callback invokers are called. Throughout this paper, we use "callsite" and "callback invoker sites" interchangeably to refer to these locations.

Magic Methods. Special methods in scripting languages that are automatically invoked in response to certain operations (e.g., `__destruct()`, `__toString()`). Table 1 provides a comprehensive overview of magic method categories, their corresponding entrance operations, and callback invokers.

Effective Callsite Coverage. A refined metric measuring the number of callback invoker calls that successfully switch execution context from the interpreter to user-defined magic methods, as opposed to simply counting all callsites reached during execution.

Callbacks can be categorized into two types: *explicit* and *implicit*. Explicit callbacks are passed directly to a function, such as passing a custom comparison function to PHP's `usort()`. Developers are typically aware of explicit callbacks and take steps to enforce invariants, reducing the likelihood of bugs. Implicit callbacks, on the other hand, are registered ahead of time and invoked later, often without the developer's explicit awareness. Examples include magic methods (e.g., PHP's `__destruct()`, `__toString()`) and event handlers (e.g., PHP's `set_output_handler()`). Because developers may not realize when or how these callbacks are invoked, implicit callbacks are more

Table 1. Categories of Magic Methods and their Callback Invokers

Category	Magic Method	Entrance Operation	Runtime Behavior	Callback Invoker(s)
Lifecycle	`__destruct`	`unset($obj)`	Cleans up resources when object is destroyed	`zval_ptr_dtor` `zend_std_object_dtor`

Type Conversion	`__toString`	`echo $obj`	Converts object to string in string context	`zend_cast_to_string`
				`zend_compare`
				`zend_std_compare_objects`

Property Access	`__get`	`$obj->property`	Handles access to inaccessible properties	`zend_std_call_getter` `zend_std_read_property`
Serialization	`__serialize`	`serialize($obj)`	Defines how an object is serialized	`php_var_serialize`
Object Cloning	`__clone`	`clone $obj`	Defines custom behavior when object is cloned	`zend_objects_clone_members` `zend_objects_clone_obj`
Function Calling	`__invoke`	`$obj($args)`	Allows object to be called as a function	`zend_init_dynamic_call_object` `zend_std_get_closure`

prone to bugs. Furthermore, to our best knowledge, no interpreter currently provides effective runtime mitigation for such issues.

3.2 Data Collection Methodology

To ground our analysis in real-world bugs, we collected 150 proof-of-concept (PoC) scripts that triggered callback bugs across three languages: 50 each for Python, PHP, and Ruby. We examined each language's issue tracker for entries involving magic methods (e.g., `__destruct()`) and retained only those accompanied by a PoC.

We selected recent PoCs from these issue trackers and reproduced them on AddressSanitizer-instrumented interpreters, counting a bug as triggered if the PoC crashed the interpreter. Similar to ground-truth benchmarks like MAGMA [18], we consider 150 PoCs to be a reasonable sample size. The PoCs include class and method definitions, unary/binary operations, and calls, with simple loops and conditionals. These observations guided the design of our targeted script generation (Section 4.2), including a simple callback-bug-focused intermediate representation that we describe in Section 4.3.

During our systematic analysis of these PoCs analysis, we observed frequently appearing keywords like "side-effects", "unexpected mutation", and "magic methods" in bug reports. When reproducing the PoCs on AddressSanitizer-instrumented interpreters, we recognized the same recurring stack trace pattern: entrance operation → callback invoker → magic method → crash. This consistent bug pattern across all three languages motivated us to formalize the underlying behavior as runtime invariants and preconditions.

To reliably attribute each observed crash back to its originating PoC (rather than double-counting when the same underlying bug was reported by multiple PoCs), we first grouped crashes by AddressSanitizer crash site and kept one representative PoC per group; we then manually inspected the crashing script to identify its magic methods and side-effect operations. This crash-site grouping is the same mechanism later used in the reproduction study (Section 5.4) to count unique reproducible bugs, but here it is applied only to the historical PoCs we collected rather than to fuzzer output.

The following subsections build on this dataset: Section 3.3 derives the five invariants that callback bugs violate and uses them to define "callback bug" formally, and Section 3.4 enumerates the four preconditions required to trigger one.

3.3 Five Invariants of Interpreter Callbacks

Based on the collected dataset, we identified *five* common assumptions interpreters make about implicit callbacks, as well as the root causes of bugs that arise when these assumptions are violated. We refer to each assumption as an *invariant* because the interpreter silently requires it to hold across every callback invocation. We report the percentage of these five invariants according to how often they are broken in the collected dataset.

Immutability (72.7%). Interpreters often assume that containers (e.g., arrays, heaps, or deques) should not be freed, resized, reallocated during callback execution. Violation of this invariant leads to memory corruptions, such as: *Use-After-Free*: when a callback frees an object or memory region that will be later accessed by the runtime; *Out-of-Bounds Read or Write*: when a callback resizes a buffer or array, but the runtime is unaware of the change; *Null-Pointer Dereference*: when a callback frees an object or sets a field to `NULL` that will be later dereferenced by the runtime.

Non-Reentrancy (8.0%). Interpreters assume that callbacks should not be re-entered. Violating this assumption leads to *Stack Overflow*: when a callback is invoked recursively, or *VM State (Memory) Corruption*: when a callback is invoked in an unexpected context (e.g. during garbage collection).

Return Type Coherency (7.3%). Interpreters assume that callbacks should return values of the expected type. Violation of this invariant results in type confusion. This is particularly problematic in scripting languages because of dynamic type checking.

Exception Safety (2.7%). Interpreters assume that callbacks should not throw exceptions. Otherwise, the runtime will continue execution in a corrupted state, leading to undefined behavior.

Assertion Failure (9.3%). In some cases, callbacks violate invariants that are explicitly asserted by the runtime, causing a crash.

These percentages represent aggregated statistics across all three languages in our dataset. The distribution is similar across Python, PHP, and Ruby, with immutability remaining the most frequently violated invariant in all three interpreters. The collected PoCs are relatively short in terms of LoC, as they have been reduced by bug reporters and developers. In most cases, they contain fewer than 20 lines. Syntactically, these PoCs primarily consist of class definitions, magic method definitions, unary/binary operations, and function/method calls, with simple loops and conditionals. These observations served as inspiration for the targeted script generation procedure described in Section 4.2, as well as the simple callback-bug-focused intermediate representation we later introduce in Section 4.3. With the five invariants in hand, we can now state formally what constitutes a callback bug.

Callback bugs occur when user code violates the interpreter's invariants, and the interpreter lacks mechanisms to prevent or mitigate these violations.

3.4 Four Preconditions of Callback Bugs

Callback bugs are challenging to detect because they depend on four **preconditions** that *must all hold* for the bug to manifest. These preconditions collectively enable the violation of runtime invariants, and the absence of any one precondition would prevent the bug from occurring. We summarize the preconditions below and illustrate them in Figure 1.

P1 *Reaching a callback*: The interpreter must invoke a callback (e.g., magic methods such as `__toString()`, `__eq()`, or `__destruct()`). This is the foundational step, as without invoking a callback, no callback-related bug can occur. In Figure 1, **P1** is satisfied when `unset($arr[0])` triggers `__destruct()` via `zval_ptr_dtor()`.

P2 *Involving user-controlled objects*: The object associated with the callback must be user-defined or user-controllable, allowing the callback to be overridden or manipulated. This precondition ensures that the callback behavior can be influenced by the attacker or user. In Figure 1, **P2** is satisfied by the user-defined class `C`, which overrides `__destruct()` to modify the global array `$arr`.

P3 *Triggering unintended side effects*: The callback must perform operations that violate runtime invariants, such as freeing memory, resizing buffers, or reallocating data objects. This precondition introduces the vulnerability by disrupting the interpreter's expected state. In Figure 1, **P3** is satisfied when `__destruct()` calls `$arr->setSize(0)`, freeing the array's memory and violating runtime invariants.

P4 *Mishandling post-callback access*: After the callback returns, the runtime must access objects or data structures that were altered by the callback, leading to undefined behavior. This precondition ensures that the side effects of the callback are exploited. In Figure 1, **P4** is satisfied as the native code attempts to set the freed array element to `NULL` after the callback returns, resulting in a use-after-free vulnerability.

These preconditions are interdependent and collectively enable callback bugs. For example, even if a callback is invoked (**P1**) and performs unintended side effects (**P3**), the bug cannot be exploited unless the runtime mishandles post-callback access (**P4**). Similarly, user-controlled objects (**P2**) are necessary to ensure that the callback behavior can be manipulated to trigger the bug.

> Callback bugs are characterized by four **necessary preconditions**: reaching a callback, involving user-controlled objects, triggering unintended side effects, and mishandling post-callback access. The absence of any one precondition prevents the bug from manifesting.

3.5 Limitations of Existing Approaches

Existing tools face challenges in detecting callback bugs because they typically focus on either the native side or the script side, but callback bugs must understand both sides and their interactions via entrance operations and magic methods.

Static Analyses. Existing static analyses primarily operate on the native side of interpreters [1, 12, 17]. While these tools can identify potential callsites (**P1**), they have limited ability to predict how user-defined code might violate interpreter invariants (**P3**). For example, static analysis might flag `zval_ptr_dtor()` as a potential callsite, but cannot determine whether the `__destruct()` method will perform harmful side effects like resizing arrays. This limitation exists because static analysis operates on native C code and lacks visibility into dynamic, user-defined script code semantics: it cannot predict what operations a magic method will perform, whether it will modify global state, or if it will violate interpreter invariants. Additionally, static analysis alone cannot generate proof-of-concepts to validate findings. Cross-language static analysis approaches like Codon [10, 22] address different problems: they analyze existing polyglot programs for library-level API misuse rather than detecting interpreter-internal callback bugs during script generation.

Fuzzers. Grammar-based fuzzers like Nautilus [9] and Gramatron [25] often struggle to satisfy **P1** and **P2** simultaneously. Their grammars may lack constructs for defining custom classes with magic methods, as introducing such constructs without proper instantiation often results in dead code. Corpus-based fuzzers like Polyglot [13] and CodeAlchemist [16] may occasionally satisfy **P1**, but often fail to meet **P2** and **P3**, as generating custom classes with overridden magic methods requires the seed corpus to already include such examples. Most fuzzers also lack semantic awareness of the invariants that callback bugs violate, making it difficult to systematically trigger vulnerabilities.

Algorithm 1 Context Link Analysis Algorithm

Input: Interpreter LLVM IR IR, Magic-method-to-invoker mapping $\mathcal{M} \rightarrow \mathcal{I}$ (manually curated; see Section 4.1), Entrance operations $\mathcal{E}$
Output: Link mappings $L_1 : \mathcal{E} \rightarrow C$ (callback invoker sites), $FilteredCallsites$
Phase 1: Build Link 1 ($\mathcal{E} \rightarrow C$)
$CallGraph \leftarrow$ ExtractCallGraph(IR)
$OpHandlers \leftarrow$ IdentifyOpcodeHandlers(IR, $\mathcal{E}$)
$\mathcal{I} \leftarrow$ LookupInvokers($\mathcal{M} \rightarrow \mathcal{I}$) ▷ invoker *functions*, from the mapping table
for each $e \in \mathcal{E}$ **do**
  $handler \leftarrow OpHandlers[e]$
  $reachable \leftarrow$ BFS($CallGraph$, $handler$, $\mathcal{I}$) ▷ callsites of $\mathcal{I}$ reachable from $handler$
  $L_1[e] \leftarrow reachable$
end for
Phase 2: Filter Callsites
$FilteredCallsites \leftarrow \emptyset$
for each $c \in C$ **do**
  if HasPostCallbackAccess(c) $\wedge$ ManipulatesObjectPointer(c) **then**
    $FilteredCallsites \leftarrow FilteredCallsites \cup \{c\}$
  end if
end for
return $L_1, FilteredCallsites$

These challenges highlight the **need for specialized techniques** to detect callback bugs effectively. Addressing this gap requires a *hybrid* approach that combines the strengths of static analysis and fuzzing to systematically detect callback bugs to fix them early.

4 Design and Implementation of CrossFit

Our approach *combines* context link analysis and targeted script generation to address the limitations of existing tools. The context link analysis identifies *callsites* by establishing a correspondence between magic methods and native functions (e.g., `__destruct()` and `zval_ptr_dtor()`), filtering for callsites where operations are performed after the callback returns (**P1**, **P4**). It also identifies *entrance operations*–builtin method or function calls that lead to magic method calls–using call graph analysis (**P1**), and *side-effect operations*–methods that modify objects or data structures–using heuristics (**P3**). Fuzzing complements this by generating proof-of-concepts (PoCs) that specifically target the links and callsites identified by context link analysis, enabling systematic validation of potential vulnerabilities. The fuzzer prioritizes generating the main program body with entrance operations (**P1**), custom classes with magic methods (**P2**), and side-effect operations within those methods (**P3**). By leveraging a minimized seed corpus for competitive callsite coverage, our approach systematically satisfies the preconditions required to trigger callback bugs, enabling their detection.

4.1 Context Link Analysis

To detect callback bugs, we must bridge the causal gap between script-level operations and interpreter runtime behavior. As systematically depicted in Algorithm 1, our context link analysis establishes two critical mappings: (1) connecting *entrance operations* to callback invoker sites, and (2) mapping those sites to their corresponding magic methods.

Each phase addresses specific challenges in cross-language analysis:

(1) **Link Construction**: LLVM call graph analysis determines which entrance operations reach which callback invoker sites, handling complex C constructs transparently at the IR level.
(2) **Callsite Filtering**: Heuristic-based elimination of callsites unlikely to trigger bugs, focusing on sites with post-callback object access patterns.

The remainder of this section details each phase, explaining how our approach handles the technical challenges raised by prior work in cross-language analysis.

Phase 1: Entrance Operations to Callback Invoker Sites. The first phase constructs Link 1 by analyzing how script-level operations eventually reach callback invoker sites in the interpreter. Table 2 illustrates this process with a concrete example from PHP, showing how `unset(obj)` eventually triggers `__destruct()`.

Table 2. Example link construction chain in PHP

Level	Component
L0: Script	`unset(obj)`
L1: Bytecode	`ZEND_FETCH_DIM_UNSET`
L2: Interpreter (C)	`ZEND_UNSET_DIM_SPEC_CV_CONST_HANDLER()` → `spl_fixedarray_object_unset_dimension()` → `execute_ex()` → `zval_ptr_dtor()`
L0': Script	`__destruct()`

This analysis operates at three abstraction levels:

- **L0 → L1**: One-time manual mapping of ≈100 PHP opcodes to their bytecode representations
- **L1 → L2**: Automated LLVM-IR call-graph analysis revealing function call chains (e.g., `ZEND_UNSET_DIM_SPEC_CV_CONST_HANDLER()` eventually calls `zval_ptr_dtor()`)
- **L2 → L0'**: One-time manual mapping of 15 PHP magic methods to their ≤4 invokers each

Two key technical challenges arise in our context link analysis: handling complex C constructs and selecting appropriate callback invokers from call chains.

Handling Complex C Constructs. Our analysis operates on LLVM IR, which naturally handles complex C features like macros, inline functions, and function pointers. Macros are expanded during compilation, inline functions are either preserved or inlined transparently, and function pointers appear as indirect calls with type signatures that we conservatively match. When callback invokers like `zval_ptr_dtor()` are implemented as macros, they become visible in the LLVM IR as their expanded forms, allowing our call graph analysis to capture them.

Invoker Selection Strategy. Selecting the *callback invokers* (the native functions whose *callsites* the BFS treats as sinks) is a one-time manual step performed by the user who is porting CrossFit to a new interpreter, not an automatic output of CrossFit itself. Concretely, for each magic method the user traces backward from the C function that dispatches the callback and picks, among the functions on that short trace, the one most appropriate to record in the mapping table. We recommend the following criteria, which a user of CrossFit applies by inspection of the interpreter source: (1) prefer the highest-level function that appears in the public, documented API (the most stable interface); (2) avoid lower-level internal helpers that are considered implementation details and may change between versions; and (3) avoid higher-level functions whose semantics are tied to a specific data structure. For the example chain `zval_ptr_dtor()` → `i_zval_ptr_dtor()` → `rc_dtor_func()` → callback, these criteria pick `zval_ptr_dtor()`: functions like `i_zval_ptr_dtor()` are internal, while higher-level wrappers like `spl_fixedarray_object_unset_dimension()` are too specific. A practical consequence is that the chosen invokers tend to appear noticeably more frequently (in terms of callsites) than other functions in the chain, which matches the frequency-based intuition mentioned in the introduction. Once the invoker functions have been selected in this way, CrossFit's BFS instruments the *callsites* of those functions, knowing that execution at any such site will eventually reach the actual callback mechanism.

The practical implementation follows the BFS algorithm outlined in Algorithm 1, lines 8-11, and proceeds forward rather than backward: it takes the user-provided set of invoker functions $\mathcal{I}$ as

already-known sinks and searches from opcode handlers and built-in object method definitions (which have distinctive function names, e.g., `SplDoublyLinkedList->serialize()` corresponds to `zim_SplDoublyLinkedList_serialize()`) toward those sinks. Because $\mathcal{I}$ is an input rather than something discovered by the BFS, there is no circularity between invoker selection and call-chain analysis: the manual selection step uses short, backward traces from each magic method, while the BFS uses the resulting $\mathcal{I}$ to find *callsites* reachable from entrance operations.

Callback Invoker Sites to Magic Methods. While the algorithm focuses on the automated analysis phases, establishing the correspondence between callback invoker functions and their associated magic methods requires manual effort for each language. Table 1 presents key examples of our mapping for PHP, showing how each magic method maps to native callback invokers. The complete correspondence table for all magic methods across PHP, Python, and Ruby is provided in our artifact [33]. This mapping is based on manual analysis of interpreter source code and represents a one-time effort for each language. The manual analysis process involves: (1) identifying all magic methods through documentation review, (2) tracing execution paths through interpreter source code to find native callback invokers. For example, in PHP, we traced `unset($obj)` through the Zend Engine source to discover it calls `zval_ptr_dtor()`, which triggers `__destruct()`. Once established, this mapping enables our fuzzer to generate targeted (entrance operation, magic method) pairs for test synthesis. The manual mapping approach is required because: (1) callback invokers are language-specific implementation details not exposed through APIs, (2) the correspondence varies between interpreter versions, and (3) automated discovery would require complex program analysis facing the same cross-language challenges we aim to solve.

Phase 2: Callsite Filtering for Post-Callback Access. The second phase implements the filtering logic from Algorithm 1 to eliminate callsites unlikely to trigger bugs. This addresses precondition **P4** by focusing on sites where post-callback access occurs. Our filtering heuristic leverages a key insight: callbacks can only manipulate script-side objects (e.g., `zval *` in PHP), not arbitrary native variables like local `int` counters. Therefore, we retain only callsites that: (1) load or store object pointers after the callback returns, or (2) pass object pointers to subsequent native functions. This simple yet effective filter eliminates many low-risk sites, particularly those on error-return paths where functions exit immediately. While a complete dynamic taint analysis offers more precision, it requires substantial engineering effort and faces the same cross-language challenges we aim to address. Our lightweight static filter achieves practical effectiveness while maintaining implementation simplicity.

Instrumentation and Coverage Metrics. With filtered callsites identified, we implement an LLVM instrumentation pass that logs callback invocation during execution, recording caller location, file, line number, and associated magic method. To enable this logging, we leverage the manual mapping between magic methods and their native invokers established during context link analysis. In total, we identified 15, 42, and 17 magic methods for PHP, Python, and Ruby respectively, along with 22, 54, 36 associated callback invokers. This enables comprehensive monitoring of callback-related execution paths during targeted script generation.

4.2 Targeted Script Generation

While static analysis identifies potential callsites, fuzzing is essential for generating PoCs and validating these findings. Figure 2 provides a high-level overview of our script generation scheme, which comprises Phases 3 and 4 (continuing from the two analysis phases in Section 4.1). In Phase 3, we generate program templates (skeletons) with class definitions containing magic methods and *entrance operations* that can reach these methods. In Phase 4, we fill magic method bodies with side-effect operations to trigger bugs.

Category	Magic Method	Entrance Operation	Runtime Behavior	Callback Invoker(s)
Lifecycle	__destruct	unset($obj)	Cleans up resources when object is destroyed	zval_ptr_dtor zend_std_object_dtor

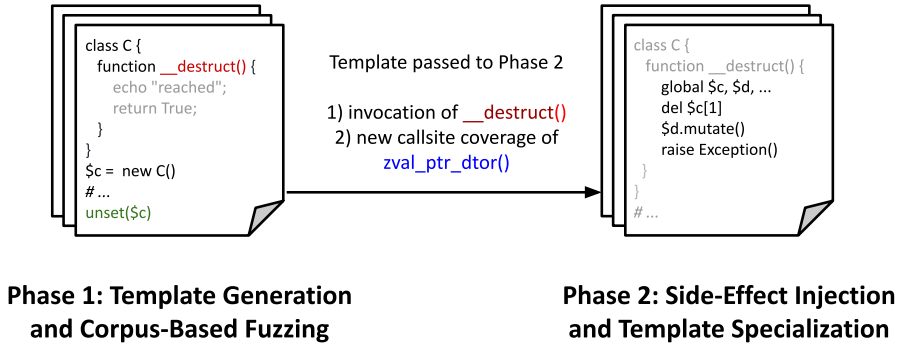

Fig. 2. Targeted script generation

Phase 3: Template Generation and Corpus-Based Fuzzing. This phase operates in iterative rounds to generate diverse program templates that satisfy preconditions **P1** and **P2**. Based on our collected dataset of 150 historical callback bugs, we observed that certain bugs require cooperation between multiple magic methods (e.g., an object must be both hashable and comparable). The maximum number of magic methods used in any historical proof-of-concept is four. Therefore, CrossFit samples 1-4 magic methods uniformly at random from the available set. Since there are only tens of magic methods in each language, and combinations of up to 4 elements have manageable search space, random sampling is sufficient and complex heuristics are not necessary.

We generate a class declaration with the sampled magic methods defined. Initially, magic method bodies contain only print statements to notify when invoked (these are replaced in Phase 4). If a magic method requires a specific return type (e.g., `__toString()` must return a string), we generate appropriate return statements with random constant values. For each selected magic method, we generate corresponding *entrance operations* based on our static analysis mapping. Due to static analysis imprecision, we compensate by generating "other operations" (i.e., the IR-based code generation detailed in Section 4.3): primarily method calls and binary operations that set up necessary context (e.g., `a.append(obj_with_del)` to populate lists).

We utilize seed corpus inputs alongside fresh generation. The corpus provides valid combinations of magic methods and *entrance operations* that have been observed to work in practice. Templates advance to Phase 4 only if they: (1) define at least one magic method that gets invoked during execution, and (2) achieve new callsite coverage. This ensures we focus on programs relevant to callback bugs rather than arbitrary code generation.

Phase 4: Side-Effect Injection and Template Specialization. In this phase, we uniformly select a template from the corpus generated in Phase 3 and specialize it by filling magic method bodies with side-effect operations to satisfy precondition **P3**. We fuzz each selected template 10 times before moving to the next template.

We first capture all variables defined in the main program body and declare them in the local scope with the `global` statement in PHP and Python, ensuring side effects can affect the broader program state.

To decide which method calls should be injected, we use a keyword-based heuristic built from the dataset of historical PoCs and from the invoker-selection process itself. Across the three languages, the methods that actually caused invariant violations in historical bug reports, as well as many of the native-side invokers we selected, share a small set of recurring English verbs: `clear`, `delete`, `resize`, `update`, and similar. We therefore assemble a language-agnostic keyword list of these verbs and bias method-call generation toward methods whose names contain them (the complete list is available in our artifact [33]). The list does not shrink CrossFit's sample space, but it increases the probability of generating calls that modify or free data structures, which in turn increases the rate at which we satisfy **P3**. Omitting the list only slows discovery of certain bugs. Beyond name-based biasing, we additionally ensure that the generated programs are syntactically and semantically correct by tracking variable definitions and their types.

Generating Terminator Statements. To test *return type coherency, non-reentrancy, and exception safety* (Section 3.3), we devise a strategy for generating or mutating terminator statements in magic methods. With equal probability, the fuzzer may generate:

(1) A return statement with the correct or wrong (equal probability) return type.
(2) A raise statement that throws a random builtin exception.
(3) An await/yield statement that suspends execution.
(4) A re-entrant call that invokes the same entrance operation, leading to another invocation of the callback.

These terminator statements ensure that the fuzzer explores a diverse set of control flow paths, increasing the likelihood of triggering bugs.

4.3 Implementation

CrossFit is implemented from scratch in 4,749 lines of Python and 471 lines of C++ to develop LLVM passes for the context link analyses. Our implementation requires three inputs: (1) the interpreter source code compiled to LLVM IR for static analysis (all three targets support compiling with LLVM link time optimization (LTO), we simply run our pass on the LTO-linked bitcode file) (2) a manually constructed CSV mapping table between magic methods and callback invokers (detailed in Section 4.1), and (3) optional seed corpora from official test suites to bootstrap fuzzing. For bug detection, we use AddressSanitizer as our oracle, which detects memory safety violations. Semantically incorrect scripts will simply cause the interpreter to exit with a non-zero code but will not crash (e.g., SIGSEGV). Therefore, all bugs reported are true positives and have been promptly fixed by developers. To understand the engineering effort required, especially for adapting CrossFit to new languages, Table 3 provides a breakdown of code across shared components and language-specific modules.

The implementation consists of two key components that enable cross-language script generation:

Program Representation. The targeted script generation phase requires a unified intermediate representation to handle multiple programming languages (PHP, Python, Ruby) systematically. Specifically, the IR-based code generation must support three capabilities used during Phases 3 and 4: (1) generating syntactically correct programs across different languages, (2) performing mutations and transformations on program structures, and (3) ensuring semantic consistency when filling magic method bodies with side-effect operations. We implement a statement-like IR where each IR instruction corresponds to a statement in Python or PHP, similar to FuzzILLI [15, 29]. The IR supports basic binary operations, typecasting, builtin function/method calls, setting object properties, and entrance operations listed in Table 1. Some if statements and loops with fixed iteration are included to trigger more diverse program behaviors, along with special control flow transfer statements to test invariants. To make use of seed corpora, we leverage `py-tree-sitter` [8]

Table 3. Lines of code breakdown for CrossFit components

Component	Shared	Python	Ruby	PHP	Total
LLVM Analysis Pass (C++)	471	-	-	-	471
IR & Script Generation (Python)	4,152	-	-	-	4,152
Lifter (Python)	-	166	222	209	597
Mapping Table (CSV)	-	111	36	22	169

to implement a parser that turns PHP and Python source code into our IR. During fuzzing, the IR program is lifted to program source and fed into the interpreter for execution. A complete list of all IR instructions can be found in our artifact [33].

Semantic Correctness. Semantic correctness has been a relatively well-studied research topic featured by [16, 32]. In essence, most language fuzzers try to ensure type correctness through specification [15, 23] or runtime introspection [2, 19]. We leverage the type specification in `typeshed` [3], `rbs` [24] and `*.stub.php` files to generate type correct statements for each language (in the main program and method body). We use reflection to determine the types of values in the seed corpus similar to PyRTFuzz [2] and SoFi [19] [34]. We also wrap each generated statement in a try-catch block to handle exceptions and avoid early termination, ensuring that the fuzzer does not waste execution cycles.

5 Evaluation

We evaluate CrossFit through four research questions to assess its effectiveness in discovering and reproducing callback bugs.

RQ1 How does CrossFit's performance in terms of callsite coverage compare to existing tools?
RQ2 How does CrossFit's context link analysis contribute to bug discovery?
RQ3 What new bugs does CrossFit discover, and what are their characteristics?
RQ4 How effectively does CrossFit reproduce old bugs?

5.1 Experimental Setup

Targets. We choose recent Python (v3.13.0), PHP (v8.4.0), and Ruby (v3.4.0), as they are the most widely used scripting languages. We config them to enable all extensions with AddressSanitizer (ASan) as our oracle to detect memory corruptions.

Comparison with Static Analysis Tools. While static analysis approaches exist for multi-language programs [10, 22], they address fundamentally different problems. They analyze API interactions between user code and external libraries, focusing on library-level API misuse in existing polyglot programs. In contrast, CrossFit detects interpreter-internal callback bugs where user-defined magic methods violate the interpreter's runtime invariants. Even sophisticated techniques like cross-language call graphs cannot address callback bug detection because they lack awareness of the interpreter's internal callback invoker chain and the specific invariants that callback bugs violate. Since no existing static analysis tools are tailored to detect callback bugs, we implement our own static analysis (Section 4.1) and conduct an ablation study (Section 5.2).

Comparison with Pure Fuzzing Tools. For a baseline, we compare against PolyGlot and Nautilus, two grammar-based fuzzers, as they are the only fuzzers we know support fuzzing all three languages: PHP, Ruby and Python. PolyGlot's grammar is adapted from ANTLR's grammar, which should be relatively complete and covers all language features. Nautilus's grammar is handcrafted. Since Nautilus does not generate any definitions of class or function, to ensure a

Table 4. Edge, callsite, and effective callsite coverage table

	Cov. Type	Polyglot	Nautilus	CrossFit	CrossFit-Link	CrossFit-Corpus	Corpus (baseline)	Total
Python	edge	44418	22787	32084	-	-	43419	154156
	callsite	2546	1214	**2722**	2594	1553	2512	4973
	effective	206	37	**446**	207	143	184	4973
PHP	edge	28520	21314	16328	-	-	26791	156400
	callsite	741	150	**811**	740	498	735	2388
	effective	144	23	**249**	146	95	121	2388
Ruby	edge	9847	15923	14521	-	-	9721	87420
	callsite	220	250	**258**	218	234	194	1456
	effective	72	24	**142**	64	112	56	1456

fair comparison, we add definitions of class and function manually to the grammar of nautilus, (around 100 grammar rules addition to all languages), likely the highest boost a knowledgeable developer and make Nautilus target specifically callback bugs. We also add a `try-catch` guard for every statement generated by Nautilus and PolyGlot to ensure programs generated by the baseline fuzzers could continue execution even if an error is thrown, similar to how CrossFit ensure semantic correctness.

Experiment configurations. We run all experiments on a cluster node of a 16-core Intel Xeon CPU machines with 64 GB of memory (with hyper-threads disabled). We obtain initial seeds for CrossFit and PolyGlot from the official unit test suite of PHP (3,812 seeds), Ruby (797 seeds) and Python (6,786 seeds). Each fuzzing instance is pinned to a single core using `cpuset`. All experiments are 24 hours and repeated eight times for each fuzzer-target pair. Bugs are reported throughout the development of CrossFit which spans one month.

5.2 Callsite Coverage Comparison (RQ1, RQ2)

Table 4 presents the coverage results for PolyGlot, Nautilus, and CrossFit across three metrics: edge coverage (using LLVM sancov instrumentation), callsite coverage, and effective callsite coverage. In total, we instrument 4,973, 2,388, and 1,456 native function callsites for Python, PHP, and Ruby, respectively, nearly two orders of magnitude fewer than the edges instrumented by sancov.

Edge Coverage Results. In terms of edge coverage, CrossFit does not show an advantage over traditional language fuzzers like PolyGlot or Nautilus. In fact, CrossFit's edge coverage decreases compared to the initial corpus. The reason is that CrossFit is designed to focus on callback bugs. To achieve this, it distilled 939 seeds from the initial 3,812 in the corpus, while the other seeds were filtered out by our context link analysis because they neither defined magic methods nor triggered new callsite coverage. In contrast, the use of edge coverage causes traditional fuzzers like Nautilus and PolyGlot to be "unfocused" and "shortsighted". For instance, Nautilus prioritizes inputs that call previously unexercised built-in functions with correct type arguments, even if these inputs do not contribute to reaching callback-related callsites. Simple constructs like class and method definitions are generated early in the campaign but rarely lead to new edge coverage later.

While general code coverage metrics (e.g., edge or block coverage) can be useful for detecting some categories of interpreter bugs, callback bugs require reaching specific *callsites* (**P1**). If a bug-triggering callsite is never reached, the bug cannot manifest. Conversely, merely reaching a generic piece of code does not imply that the necessary preconditions for a callback bug (**P2**–**P4**) are met. Consequently, *callsite coverage* is a more accurate measure of progress toward finding callback bugs.

Callsite Coverage Results. Compared to PolyGlot, CrossFit improves callsite coverage for Python, PHP, and Ruby by 8.36%, 10.34%, and 32.99%, while PolyGlot achieves only 1.35%, 0.82%, and 13.40% improvement. On average, CrossFit outperforms PolyGlot by **12.04%**. To ensure a fair comparison with Nautilus, which does not use an initial seed corpus, we also evaluate CrossFit without the initial corpus, denoted as CrossFit-NoCorpus. Starting from scratch, CrossFit-NoCorpus achieves significantly higher callsite coverage than Nautilus by **1.85x**.

Our evaluation shows that the corpus raises callsite coverage from 1,553 (CrossFit-NoCorpus) to 2,722 (CrossFit), demonstrating the effectiveness of our design. Using corpus seeds alongside fresh generation provides valid combinations of magic methods and entrance operations that are known to work in practice. However, the corpus is not the primary factor in CrossFit's effectiveness. Importantly, CrossFit-NoCorpus still found 6 of the 12 bugs detected by the full CrossFit, while PolyGlot (even with the same corpus) found nothing. This demonstrates that CrossFit's context link analysis and targeted generation are the key differentiators, not merely the corpus.

Effective Callsite Coverage Results. The gap becomes even more pronounced when examining *effective callsite coverage*. PolyGlot and Nautilus achieve effective callsite coverage of only 206 and 37 for Python, 144 and 23 for PHP, and 72 and 24 for Ruby, respectively. In contrast, CrossFit achieves 446 for Python, 249 for PHP, and 142 for Ruby, representing a dramatic improvement of **1.96x** on average over PolyGlot. This striking difference occurs because while baseline fuzzers may occasionally reach callback invoker sites (e.g., through `unset(new stdClass())`), they rarely generate code that passes user-defined objects with overridden magic methods to these sites. CrossFit systematically addresses this by generating *related entrance operations* for magic methods, leveraging relationships inferred during static analysis. However, CrossFit does not achieve full effective callsite coverage compared to the regular callsite coverage, as the language runtime will often create temporary objects and call them with callback invoker functions, but since these objects are not controllable by the user, they do not have user-defined magic methods and thus do not contribute toward effective callsite coverage.

In contrast, CrossFit adopts a targeted approach. Once a callsite is reached, CrossFit focuses on generating method bodies without relying on coverage guidance, as traditional coverage metrics can dilute the fuzzer's attention toward inputs that increase edge coverage but fail to satisfy the four preconditions for callback bugs. This targeted strategy allows CrossFit to maintain a strong focus on callback-related code paths.

During the coverage campaign, neither PolyGlot nor Nautilus found any callback bugs. PolyGlot's *constrained mutation* is limited to inserting new statements, mutating operands in expressions, and renaming variables, but it does not generate new classes (even though its grammar supports such constructs). Nautilus, on the other hand, suffers from semantic correctness issues, as it generates program statements purely from a context-free grammar, often resulting in variable reference errors and type errors. These limitations highlight the importance of CrossFit, combining context link analysis and targeted script generation to systematically satisfy all preconditions for callback bugs detection.

A more meaningful approximation metric for detecting callback bugs is effective callsite coverage, which measures callsites where user-defined magic methods are actually invoked. CrossFit dramatically outperforms existing language fuzzers in this fine-grained metric by **1.96x** on average, while callsite coverage shows more modest improvements of **12.04%** (with corpus) and **1.85x** (without corpus).

Ablation Study. To understand why CrossFit outperformed the baseline fuzzers, we conduct an ablation study comparing CrossFit without the information from relationship inference analyses and randomly generates operations in the first phase of fuzzing, denoted as CrossFit-Link. CrossFit-Link achieved significantly lower callsite coverage, performing similarly to PolyGlot. In total, there are 8,118 builtin functions, every one of which can be an entrance function. Randomly generating these function / method calls is unlikely to lead to invoking the magic callback. In terms of bug found, CrossFit-NoCorpus detected 6 of the 12 php bugs detected by CrossFit at least once in one of the 8 x 24 hour campaigns. CrossFit-Link did not detect any bugs.

It is worth noting that traditional fuzzers like PolyGlot and Nautilus can satisfy individual preconditions (e.g., generating class definitions or invoking built-in functions) thanks to their grammars. However, without awareness of the relationships between these preconditions, they often fail to satisfy all of them simultaneously. This explains why PolyGlot and CrossFit-Link achieved similar callsite coverage but fail to discover new callback bugs. The "almost equal coverage but no new bugs" phenomenon underscores that finding callback bugs is not just about coverage but also about systematically satisfying all prerequisites.

Our ablation study reveals that callsite coverage alone may not fully capture the effectiveness of callback bug detection tools. The disconnect between CrossFit-Link's callsite coverage (similar to PolyGlot) and its bug-finding capability (zero bugs found) demonstrates that simply reaching callback invoker sites is not sufficient. Callsite coverage counts all callsites reached during execution, regardless of whether they actually invoke user-defined magic methods with meaningful side effects. For example, `unset(new stdClass())` reaches the `zval_ptr_dtor()` callsite but does not invoke a user-defined `__destruct()` method. This can lead to inflated coverage numbers that do not correlate with actual bug discovery effectiveness. In contrast, effective callsite coverage specifically measures callsites where user-defined magic methods are actually invoked, providing a more fine-grained and accurate assessment of a tool's ability to trigger callback bugs.

The importance of effective callsite coverage becomes evident when comparing CrossFit with CrossFit-Link. While both versions may achieve similar raw callsite coverage, their effective callsite coverage differs dramatically. CrossFit's context link analysis ensures that when a callback invoker site is reached, it is more likely to be invoked with user-defined objects that have overridden magic methods. This targeted approach results in higher effective callsite coverage, which correlates strongly with bug discovery capability. CrossFit-Link, lacking this systematic approach, may reach many callsites but fails to trigger the user-defined callbacks that expose vulnerabilities. This distinction explains why effective callsite coverage serves as a better predictor of a fuzzer's ability to find callback bugs than traditional coverage metrics.

The context link analysis enable CrossFit's *two-side awareness* by: (i) using static analysis to pinpoint which builtin script operations serve as *entrance operations* and (ii) explicitly generating those script operations, along with custom classes and side-effect instructions to trigger deeper call chains in the native code.

Context link analysis is an integral part of CrossFit's design, without it CrossFit may satisfy preconditions individually but not at the same time, leading to significantly fewer effective callsites reached and bugs discovered. The dramatic difference in effective callsite coverage between CrossFit and baseline tools demonstrates the importance of systematic precondition satisfaction for callback bug detection.

Table 5. Bugs discovered by CrossFit.

language	issue	callback	crash site	type
php	#16649	__destruct	php_splice	uaf
	#16648	__tostring	zend_std_compare_objects	uaf
	#16646	__destruct	spl_array_unset_dimension	uaf
	#16592	__serialize	zif_msg_send	nptr
	#16591	__serialize	php_check_shm_data	assert
	#16590	__serialize	ps_srlzr_encode_php	uaf
	#16589	__serialize	zim_SplDoublyLinkedList_serialize	uaf
	#16588	__serialize	zim_SplObjectStorage_serialize	uaf
	#16479	__destruct	zim_SplObjectStorage_setInfo	uaf
	#16478	__destruct	spl_fixedarray_object_unset_dimension	uaf
	#16464	__destruct	zim_SplDoublyLinkedList_offsetSet	uaf
	#16337	__tostring	spl_ptr_heap_destroy	uaf
ruby	#21303	hash	rb_ary_difference_multi	oob
	#21304	hash	rb_ary_hash_values	uaf
	#21305	hash	set_merge_enum_into	uaf
	#21306	block	set_i_initialize	uaf
	#21331	block	st_general_foreach	uaf
	#21332	eql?	set_general_foreach	uaf
	#21333	block	rb_st_update	uaf
python	#119004	__eq__	_odict_keys_equal	uaf

5.3 Newly Found Bugs and Case Studies (RQ3)

Using our approach, we found in total **20** new bugs (Table 5) in the PHP, Ruby and Python. Many were introduced years ago but had gone undetected because no tool or test harness systematically generated the script-level constructs needed to invoke these callbacks. All bugs have been fixed and confirmed by the developers. Below, we present two in-depth case studies detailing CrossFit discovery process and explaining why the two competitor fuzzers failed to reproduce these bugs.

Use-after-free in PHP's asort shown in Figure 3a. Through context link analysis, CrossFit first discovered that `asort` will eventually call `zend_compare()`. During fuzzing, CrossFit generates a class declaration with the `__toString()` magic method declared but leaves its body empty at first. It then proceeds to create the main program, prioritizing the generation of statements identified as *entrance operations* for `__toString()`. Ultimately, CrossFit finds that passing an associative PHP array with a custom object to `asort()` leads the interpreter to call `__toString()`. Once this new callsite coverage is reached, CrossFit updates the `__toString()` body to introduce side effects, capturing the associative array `$a` from the main program and inserting multiple elements. This triggers the runtime to resize the buffer used by `$a`, freeing the old buffer and copying elements into a new one. However, PHP's internal sorting algorithm does not handle reallocation properly and continues using the freed buffer, resulting in a use-after-free (**P4**).

A notable detail of this bug is that a related function, `usort`, *explicitly* accepts a user function as a comparison callback. We looked up the native code defining `usort` and discovered the developers set a flag that "locks" the array, preventing it from any mutation throughout the entire duration of the sort. However, they did not realize `zend_compare()` will invoke `__toString()` if one side is an object and the other is a string, i.e., an *implicit* type coercion, which CrossFit discovered through static analysis. This example confirms our earlier observation that *implicit callbacks* are harder to detect and highlights CrossFit ability to uncover *latent* bugs that evade manual code reviews performed by highly experienced developers.

Use-after-free in Python's OrderedDict shown in Figure 3b. `OrderedDict` in Python is internally implemented as normal python `dict` paired with a linkedlist to record insertion order. The interesting thing here is that `__eq__()` has to be invoked twice in order for this bug to be triggered.

```php
<?php
class C {
    function __tostring() {
        global $arr;
        for ($i = 0; $i < 10; $i++) {
            $arr[$i] = $i;
        }

        return "3";
    }
}

$arr = ["a" => "1", "3" => new C, "2" => "2"];
asort($arr);
```

(a) Use-after-free in PHP's `asort`

```python
import collections

class C():
    def __eq__(self, other):
        global left
        if rand(): left.clear()

    def __hash__(self):
        return 3

left = collections.OrderedDict({C(): 4, 5: 6})
right = collections.OrderedDict({C(): 4, 5: 6})
left == right
```

(b) Use-after-free in Python's `OrderedDict`

Fig. 3. Reduced PoC of crashes discovered by CrossFit

And the call to `__eq__()` must clear the `OrderedDict` on its second invocation. The native runtime first checks equality of the internal `dict`, if we clear the `OrderedDict` on this first invocation of `__eq__()`, it will also clear the internal dict. Clearing a `dict` during comparison was a bug reported by someone else in 2016 and fixed in commit `76dfb09`, almost nine years ago. So trying to trigger the same bug now is not possible. However, after the `dict` comparison, the native runtime will then traverse the internal linked list to ensure the order of the dict elements are the same. This time, if we tried to clear the `OrderedDict` it will lead to the native runtime freeing the internal linked list. And then when dereferencing the `next` node of the internal linked list, it will cause a use after free.

Both the `asort` and `OrderedDict` bugs demonstrate how easily callback bugs can slip through manual inspections and how deeply they may be hidden in the interpreter's internal operations. These issues also show the importance of **(1)** identifying specific native callsites via static analysis, **(2)** generating script-level *entrance operations* that reach those callsites, and **(3)** systematically injecting side effects in the body of magic methods to stress-test the runtime's handling of reallocation and state changes.

> CrossFit's design allowed it to discover and isolate these vulnerabilities by first achieving callsite coverage and then adding the necessary side effects to trigger the bugs. Without carefully linking *entrance operations* to callback invokers or incorporating targeted side effects, these scenarios would likely never arise in typical fuzzing or manual testing.

5.4 Bug Reproduction Study (RQ4)

We perform the bug reproduction study on Python 3.0, PHP 8.0.0, and Ruby 2.0.0, with major version same as the developing branch but also the oldest version to capture most of the reproducible bugs. We collected 50 callback bug proof-of-concepts each for Python, PHP, and Ruby (150 total). The proof-of-concepts are first ran on an AddressSanitizer built binary to check if they reproducibly crashes the interpreter. In our case 111 out of 150 collected PoCs crashes the three interpreters. After deduplicating crashes by unique crash sites, we identified 82 unique reproducible bugs in the dataset (each mapped to a distinct callback invoker site). Then we ran 8 x 24 hour campaigns for CrossFit similar to the coverage experiment but this time for older versions of PHP, Python, and Ruby. Due to the random nature of fuzzing, a bug is considered reproduced if the fuzzer creates a crashing PoC at least once out of eight runs. For this study, we collect the crashing inputs found by CrossFit and triage them manually.

Table 6. Bug reproduction study

	Collected	Reproducible	Reproduced	Percentage	Average Time
Python	50	24	16	66.67%	19min23s
PHP	50	37	29	78.38%	11min41s
Ruby	50	21	15	71.43%	25min39s

Table 6 showcase the number of reproduced bugs by CrossFit. In total, CrossFit reproduced 73.2% (60 out of 82) reproducible bugs in the dataset. Interestingly all bugs are triggered very quickly or not triggered at all. On average it takes CrossFit less than 30 minutes to reproduce a bug, and all bugs reproduced by CrossFit are triggered within 2 hours. Running longer campaigns did not lead to CrossFit reproducing more bugs. The short discovery time highlights the effectiveness of CrossFit's approach. Not all bugs were found by CrossFit. For example, certain language features like decorators are not generated by CrossFit (and are discarded when parsed from the corpus); adding these features would be a modest engineering effort.

We believe the collected dataset itself will be a good primer for developers and security researchers looking for similar bugs. The dataset is provided alongside the fuzzer in the artifact link [33]. The dataset indeed shows that callback bugs are a common occurrence in scripting languages.

CrossFit reproduced 73.2% of reproducible bugs in our dataset, with an average discovery time of less than 30 minutes. The short discovery time highlights CrossFit's efficiency in reproducing older callback bugs.

6 Threats to Validity

Internal Threat. The primary internal threat lies in the soundness and completeness of our approach. Static analysis can be an over-approximation. While our context link analysis identifies over 2,000 callsites across both PHP and Python, not all of these callsites contain actual bugs. Flagging all callsites as potential bugs would result in an unacceptably high false positive rate, and manually verifying each one is not scalable. On the other hand, fuzzing is inherently an under-approximation. However, by targeting the callsites identified by our static analysis, we focus our fuzzing efforts on high-potential areas, significantly increasing the time spent exercising each callsite and reducing the risk of false negatives. Our goal is to bring these two line over- and under-approximation closer together, balancing precision and recall.

External Threat. The main threat to external validity concerns the generalizability of our findings. Our evaluation is limited to three popular scripting languages, PHP, Ruby and Python. While CrossFit successfully discovered numerous bugs in the latest versions of these interpreters and efficiently reproduced previously known bugs, the extent to which our findings apply to other languages or environments remains an open question. Further evaluation across a broader range of languages and systems would strengthen the external validity of our approach.

Manual Effort for New Languages and Versions. A key practical concern is the effort required to extend CrossFit to new languages or adapt it to new versions of existing languages. We address each scenario separately.

Extending to New Languages. Establishing the mapping between each language's magic methods and their native callback-invokers is language-specific, but, as detailed in Section 4.1, it is a one-time manual effort. As shown in Table 3, extending CrossFit to a new language requires two main

manual components: (1) a mapping table of manually collected invoker and magic-method pairs, averaging about 56 pairs per language (169 in total across the three languages we currently support), and (2) a per-language lifter averaging about 199 LoC to handle syntactic differences.

CrossFit's per-language portion therefore averages around 255 LoC per language, which remains relatively small compared to the shared components (LLVM analysis pass and IR & script generation), totalling 4,623 LoC. Conservatively, we expect a developer to take 1-2 person-days to enumerate magic methods, grep for such names in the interpreter source code, and select callback invokers based on the criteria listed. The lifter simply converts IR statements into string forms and could be adapted with minor effort, considering the minor syntax differences (e.g., `if (cond) { body }` to `if cond: INDENT body DEDENT`).

Adapting to New Versions. For new versions of an already supported language, CrossFit typically needs no changes. For example, the latest PHP 9.0 adds no new magic methods, and existing callback invokers remain stable. New code may introduce additional callsites of existing invokers, but developers only need to rerun the analysis to find new entrance operations that reach those callsites and fuzz them. The required effort is small and one-time per language.

7 Related Work

Static Analyses. Static analysis tools have been widely used to detect bugs in interpreters and compilers. For example, *CGSAN* [17] employs intra-procedural static symbolic taint analysis to identify use-after-compacting-garbage-collection bugs, while *CID* [26] uses two-dimensional consistency checking to detect refcount bugs in the Linux kernel. Brown et al. [11] discovered use-after-free bugs caused by garbage collection in JavaScript bindings through pattern matching, and *Sys* [12] provides a framework that detects bugs via static analysis and verifies them using symbolic execution. Multi-language static analysis approaches like Monat et al. [22] and cross-language call graph construction [10] focus on API interactions between existing user code and external libraries, while compilation approaches like Codon convert Python to LLVM IR for optimization. However, these tools are not well-suited for detecting callback bugs, as they operate on the native side and cannot predict how user-defined code might violate runtime invariants. Additionally, they cannot generate proof-of-concepts (PoCs) to validate findings, making them orthogonal to our work.

Interpreter Fuzzing. Fuzzing has been a popular approach for testing interpreters and compilers. Tools like *Nautilus*, *Gramatron*, and *Polyglot* [9, 13, 25] generate scripting language code using context-free grammars. *LangFuzz* [20] and *CodeAlchemist* [16] improve syntactic correctness by combining fragments of valid code, while *Skyfire* [28] learns context-sensitive grammar from existing code samples. *Montage* [21] uses neural networks to guide test case generation, and *FreeDOM* [31] targets specific components like browser DOM trees. However, these techniques often overlook challenges posed by implicit callbacks. Our work addresses this gap by combining static analysis and fuzzing to identify under-tested callsites, discovering **20** new callback bugs.

Studies Characterizing Interpreter Bugs. Several studies have sought to characterize bug patterns in interpreters and compilers. The *CISB* [30] paper collected a set of compiler-introduced security bugs (CISB) and analyzed their root causes, highlighting the unrealistic expectation that compiler users will understand and comply with compiler assumptions. Similarly, *PyDYPE* [14] investigated dynamic typing-related practices in Python, finding that misuse of dynamic typing often leads to underlying bugs and increases maintenance efforts. These findings align with our observations about callback bugs, where violations of runtime invariants due to dynamic behavior can lead to severe vulnerabilities.

8 Conclusion

Scripting languages are widely used due to their flexibility, and the security of their engines serves as the trust base for applications. Callback bugs, a pervasive class of vulnerabilities in interpreter engines, can lead to arbitrary code execution or even sandbox escapes. In this paper, we address the challenge of detecting *callback bugs* by proposing CrossFit, a novel two-tier approach combining context link analysis and targeted script generation. Our context link analysis identifies callsites and entrance operations, while our fuzzer generates proof-of-concept triggers by defining custom classes with magic methods and introducing side effects. We discovered 20 bugs in PHP, Ruby and Python, many of which are severe use-after-free memory corruptions. Our work systematically characterizes callback bugs, provides an automated detection tool, and contributes a comprehensive benchmark, advancing the security of scripting language interpreters.

9 Data Availability

We fully support FSE's open science policy and release the source code, scripts, instructions required to run CrossFit, and intermediate experiment data publicly [33]. The artifact is available on GitHub at https://github.com/HexHive/crossfit-artifact.

Acknowledgments

We thank Qinying Wang, the rest of the HexHive crew, and the anonymous reviewers for their detailed feedback. This work was supported, in part, by SNSF PCEGP2 186974 and SNSF 200021-236559.

References

[1] [n. d.]. Clang Static Analyzer. https://clang-analyzer.llvm.org/.
[2] [n. d.]. PyRTFuzz/Apispec/PySpec/StcSpec/Pyspec/Apispec_openai_generator.Py at Master · Awen-Li/PyRTFuzz. https://github.com/awen-li/PyRTFuzz/blob/master/apispec/PySpec/StcSpec/pyspec/apispec_openai_generator.py.
[3] [n. d.]. Python/Typeshed: Collection of Library Stubs for Python, with Static Types. https://github.com/python/typeshed.
[4] [n. d.]. Tunz/Js-Vuln-Db: A Collection of JavaScript Engine CVEs with PoCs. https://github.com/tunz/js-vuln-db.
[5] [n. d.]. ZDI-16-343. https://zerodayinitiative.com.
[6] 2019. PHP Remote Code Execution Vulnerability (CVE-2019-11043). https://blog.qualys.com/product-tech/2019/10/30/php-remote-code-execution-vulnerability-cve-2019-11043.
[7] 2023. TIOBE Index. https://www.tiobe.com/tiobe-index/.
[8] 2025. Tree-Sitter/Py-Tree-Sitter. tree-sitter.
[9] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. doi:10.14722/ndss.2019.23412
[10] Anonymous Authors. 2023. Cross-Language Call Graph Construction Supporting Different Host Languages. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 184–195. doi:10.1109/SANER56733.2023.00024
[11] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*. 559–578. doi:10.1109/SP.2017.68
[12] Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In *29th USENIX Security Symposium (USENIX Security 20)*. 199–216.
[13] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *2021 IEEE Symposium on Security and Privacy (SP)*. 642–658. doi:10.1109/SP40001.2021.00071
[14] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. 2020. An Empirical Study on Dynamic Typing Related Practices in Python Systems. In *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*. 83–93. doi:10.1145/3387904.3389253

[15] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society.

[16] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

[17] HyungSeok Han, Andrew Wesie, and Brian Pak. 2021. Precise and Scalable Detection of Use-after-Compacting-Garbage-Collection Bugs. In *30th USENIX Security Symposium (USENIX Security 21)*. 2059–2074.

[18] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Nov. 2020), 29 pages. doi:10.1145/3428334

[19] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. 2021. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event Republic of Korea, 2229–2242. doi:10.1145/3460120.3484823

[20] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 445–458.

[21] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2613–2630.

[22] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis (Lecture Notes in Computer Science)*. Springer International Publishing, Cham, 323–345. doi:10.1007/978-3-030-88806-0_16

[23] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1629–1642. doi:10.1109/SP40000.2020.00067

[24] Ruby Core Team. 2023. RBS: Ruby Signature. https://github.com/ruby/rbs. Ruby type signature language and tools.

[25] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective Grammar-Aware Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Denmark, 244–256. doi:10.1145/3460319.3464814

[26] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. 2021. Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking. In *30th USENIX Security Symposium (USENIX Security 21)*. 2471–2488.

[27] SSD Secure Disclosure technical team. 2020. SSD Advisory – PHP SplDoublyLinkedList UAF Sandbox Escape. https://ssd-disclosure.com/ssd-advisory-php-spldoublylinkedlist-uaf-sandbox-escape/.

[28] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 579–594. doi:10.1109/SP.2017.23

[29] Haoran Xu, Zhiyuan Jiang, Yongjun Wang, Shuhui Fan, Shenglin Xu, Peidai Xie, Shaojing Fu, and Mathias Payer. 2024. Fuzzing JavaScript Engines with a Graph-based IR. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, New York, NY, USA, 3734–3748. doi:10.1145/3658644.3690336

[30] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3655–3672.

[31] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. Association for Computing Machinery, New York, NY, USA, 971–986. doi:10.1145/3372297.3423340

[32] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *ACM SIGPLAN Notices* 46, 6 (June 2011), 283–294. doi:10.1145/1993316.1993532

[33] Chibin Zhang. 2026. *Artifact for FSE26 CrossFit: Demystifying VM Callback Bugs in Interpreters*. doi:10.5281/zenodo.19732538

[34] Chibin Zhang, Gwangmu Lee, Qiang Liu, and Mathias Payer. 2025. REFLECTA: Reflection-based Scalable and Semantic Scripting Language Fuzzing. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security (ASIA CCS '25)*. Association for Computing Machinery, New York, NY, USA, 1772–1787. doi:10.1145/3708821.3710818

Received 2026-01-19; accepted 2026-03-24