REFLECTA: Reflection-based Scalable and Semantic Scripting Language Fuzzing

Chibin Zhang
EPFL
Lausanne, Switzerland
chibin.zhang@epfl.ch

Gwangmu Lee
EPFL
Lausanne, Switzerland
gwangmu.lee@epfl.ch

Qiang Liu
EPFL
Lausanne, Switzerland
cyruscyliu@gmail.com

Mathias Payer
EPFL
Lausanne, Switzerland
mathias.payer@nebelwelt.net

Abstract

Scripting languages such as Python and JavaScript have revolutionized modern software development thanks to their flexibility and rich functionalities. However, scripting languages provide a large attack surface, allowing adversaries to exploit bugs in the execution engines to perform sandbox escapes or execute arbitrary code. While fuzzing successfully revealed vulnerabilities in execution engines, current techniques still face scalability and semantic correctness challenges. Specifically, existing approaches fail to scale to multiple scripting languages and often lack semantic correctness.

Reflecta, our novel scripting language fuzzer, relies solely on a common introspection feature in programming languages, namely *reflection*, enabling a generic fuzzer design across different scripting languages. With reflection, Reflecta gains the capabilities to explore the rich set of language features dynamically, significantly reducing manual efforts. Reflecta thus manages to generate language-feature-rich programs and perform type-aware mutation, producing programs with high semantic correctness. We implemented Reflecta to fuzz six execution engines for four prevalent scripting languages, Python, JavaScript, Ruby, and PHP. Reflecta achieves 1.74x ~ 3.35x improvement in semantic correctness and 1.63x ~ 2.21x improvement in code coverage compared to state-of-the-art language-general fuzzers and favorably compares to manually-augmented language-specific fuzzers without any prior semantic information. Moreover, Reflecta has discovered 25 unknown bugs confirmed by the developers of PHP, MRuby, and MicroPython, 16 of which have already been fixed.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

Software Security, Fuzzing, Interpreters, Programming Languages

ACM Reference Format:
Chibin Zhang, Gwangmu Lee, Qiang Liu, and Mathias Payer. 2025. REFLECTA: Reflection-based Scalable and Semantic Scripting Language Fuzzing. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3708821.3710818

1 Introduction

Scripting languages such as Python [18], JavaScript [4], PHP [11], Ruby [15] are at the forefront of modern software development. These languages offer flexible usability and comprehensive standard libraries that facilitate rapid and adaptive development processes. Specifically, scripting language execution engines turn script source code into bytecode, execute them either through interpretation or Just-in-Time (JIT) compilation, and through their standard libraries present rich functionalities. While the execution engines of scripting languages aim to provide fundamental security measures, including memory safety and sandboxing, vulnerabilities within the execution engines themselves can jeopardize these protections, exposing the host system to risks such as sandbox escapes [31] and unauthorized remote code execution [2].

In response to these challenges, researchers have turned to fuzzing as a method to discover bugs in scripting language execution engines. However, existing approaches suffer from critical limitations that hinder widespread adoption or result in incomplete bug finding. First, they lack *scalability* to support rich language features across multiple scripting languages and implementations. Scripting languages offer stable features such as control flow, object-oriented programming, and dynamic typing, alongside frequently changing elements like APIs from standard libraries. Developing specialized fuzzers for individual execution engines is time-consuming and difficult to scale. Most existing work [46, 54], like Fuzzilli [34] (~45K LoC), focuses on JavaScript engines, particularly for JIT-related issues. While effective for JIT bugs, these techniques are less relevant for other languages and interpreters, where rich functionality is typically embodied in the standard library. Furthermore, many scripting languages have alternative engine implementations (e.g., CPython or MicroPython for Python, CRuby or MRuby for Ruby [5, 9, 19, 20]), each with modifications tailored to specific use cases. As languages continue to evolve, introducing new features and APIs, fuzzing tools that rely heavily on manual grammar adjustments or predefined corpora [21, 37, 50] become increasingly

	Nautilus	PolyGlot	Fuzzilli	SoFi	PyRTFuzz	Reflecta
Minimal spec	✗	✗	✗	✗	✗	✓
No initial corpus	✓	✗	✓	✗	✓	✓
Multiple languages	✓	✓	✗	✗	✗	✓
Semantic-aware	✗	△	✓	✓	✓	✓

Table 1: Qualitative comparison of Reflecta with state-of-the-art scripting language fuzzers. △: Semantic support of PolyGlot remains "under development" as of this writing.

unsustainable and prone to error. This limits their ability to scale for bug identification as new languages and features emerge.

Second, existing fuzzers struggle to ensure *semantic correctness* of the mutated programs. Approaches that only focus on syntactic correctness [22, 24, 39] may pass simple syntax checks (e.g., using correct operators) but likely fail the following semantic checks (e.g., type correctness). Approaches resorting to manual annotations [29, 34] are not scalable. For instance, Fuzzilli dedicates 1,182 lines just for specifying types of built-in objects and functions, plus several hundred lines of specialized profile for each individual JavaScript engine. More importantly, statically encoding the semantic information of scripting languages is not sufficient [38, 45, 46] as the dynamic nature of scripting languages dictates valid program semantics to be determined at runtime.

To navigate these challenges, we leverage *reflection*, an intrinsic and common introspection feature of modern programming languages. Reflection provides a universal way to access detailed and precise semantic information about language features at runtime, allowing for on-the-fly inspection and interaction with the language's semantics. For instance, a program may retrieve a list of available language features (e.g., built-in objects) within the current execution environment, and further query the methods, the attributes, and the type associated with a certain runtime object.

Our proposed fuzzer, Reflecta, harnesses reflection to ease the design of scripting language fuzzing and automate the fuzzing process. Specifically, Reflecta first enumerates built-in language features (i.e., built-in objects, functions, and modules) using reflection and then generates language-feature-rich programs. Reflecta also ensures the semantic correctness of mutated programs with reflection by automatically deducing the type-correct method signatures. In the end, Reflecta achieves a substantial coverage of language features while ensuring semantic correctness in the mutated programs, with only a minimal, invariable core syntax of a target language. Table 1 shows the qualitative benefits of Reflecta compared to the state-of-the-art scripting language fuzzers.

In our evaluation on CPython, MicroPython, CRuby, MRuby, PHP, and V8 Reflecta improves the program correctness by 1.74x over Nautilus and by 3.35x over PolyGlot. Furthermore, on average, Reflecta improves code coverage by 1.63x over Nautilus and 2.21x over PolyGlot. Reflecta even reaches comparable code coverage compared to heavily hand-tuned targets, V8 for Fuzzilli with manual semantic annotations and PHP for PolyGlot with an extensive pre-selected initial corpus, *without any prior semantic information*. Lastly, Reflecta discovered 25 unique unknown bugs across PHP, MRuby, and MicroPython, a majority of which missed by existing fuzzers, and 16 bugs have been fixed.

The core contributions of Reflecta are as follows:

```
# Show globally defined constants and modules.
irb> Object.constants
[:Exception, :Math, :Set, :RUBY_PATCHLEVEL, ...]

# Get type of a value.
irb> RUBY_PATCHLEVEL.class().name()
=> "Integer"

# Get the methods of an Integer object.
irb> RUBY_PATCHLEVEL.methods()
=> [:anybits?, :nobits?, :downto, ...]

# Get the arity of Integer.downto.
irb> RUBY_PATCHLEVEL.method(:downto).arity()
=> 1
```

Figure 1: Reflection in the Ruby interactive shell (irb).

- Based on the observation that reflection exposes rich language features, we present a *scalable* and *semantic-correct* approach for scripting language fuzzing, eliminating extensive manual efforts.
- With extensive evaluation, we found 25 new bugs in popular scripting language engines including: PHP, MRuby, and MicroPython. All the bugs are confirmed by the developers and 16 bugs have been fixed.
- We open source our fuzzer: https://github.com/HexHive/Reflecta

2 Background

Here, we note the key characteristics of scripting languages and discuss the implications in testing their execution engines.

Dynamic type system. Unlike statically typed languages, scripting languages assign types to runtime objects rather than variables, postponing type verification to runtime. This dynamic typing facilitates flexible development and swift prototyping, allowing, for instance, methods to return different types of data depending on the context. However, this flexibility complicates static analysis, making it challenging to ascertain the type correctness of a program or generate type-appropriate programs without actual execution.

Rich standard library. The standard libraries of scripting languages offer extensive high-level functionalities that extend well beyond basic computation, but this broad feature set increases the likelihood of bugs within the larger library codebase compared to other components (e.g., parsers) in the execution engine.

Reflection. Scripting languages commonly provide *reflection* [14], a powerful feature that enables programs to introspect the execution environment and the language structure (e.g., available modules and object types). Programs may leverage such meta-programming information to, for example, inspect the execution environment and the objects in themselves. We summarized the representative meta-programming information available through reflection as follows.

- **Available global symbols.** Available global modules, objects, and functions in the current execution environment, including the standard libraries.
- **Object attributes and type.** Available properties (i.e., data fields) and methods (i.e., member functions) of a given object, as well as the type itself.
- **Function or method arity.** The number of arguments required for a function or method.

Example of reflection. As shown in Figure 1, it only requires a few method calls to obtain semantic information using reflection. Line 2 first queries a list of language features exposed as globally-defined modules and constants, where `RUBY_PATCHLEVEL` is a global constant. To obtain `RUBY_PATCHLEVEL`'s type, line 6 invokes `class().name()` that returns its definition class, `Integer`. Similarly, to get a list of callable methods of `RUBY_PATCHLEVEL`, line 10 calls its `methods()` method. Finally, line 14 calls the `arity()` method to identify the number of accepted arguments of the method `downto()`.

Extra benefits of reflection. Reflection obtains comprehensive language features, including not only meta-programming information such as global symbols, object attributes and type, function or method arity, but also fundamental functionalities such as scope constructs and basic operators. For instance, the `with` block in Python invokes the magic methods `__enter__()` and `__exit__()`. Interestingly, all operators in Ruby are sugar-coated calls to their respective methods (e.g. `a + 1` is actually `a.+(1)`). Reflection allows to enumerate and invoke all reachable magic methods, offering a direct pathway to interact with these rich language features. Therefore, relying solely on reflection makes REFLECTA cover a substantial portion of the language features, enabling the fuzzing of interpreter, standard libraries, and the interaction between them. Additionally, unlike other approaches, the reflection-based method requires no adjustments for new scripting language features.

Threat model. Consistent with prior work [12, 21, 29], our threat model assumes the attacker can execute arbitrary script code to exploit interpreter engine bugs to obtain arbitrary native code execution. Note that script code commonly runs in a restricted sandbox enforced by the language runtime. Our research therefore aims at identifying vulnerabilities in interpreter engines to prevent potential attacks. For example, PHP attacks may use deserialization and command injection to execute arbitrary PHP code as an initial vector, and then exploit PHP engine bugs to bypass the `disable_functions` sandbox to escalate to native code execution [1]. Recently, attacks targeted the Python LLM code interpreter, with adversaries exploiting CPython engine bugs for arbitrary server code execution [56].

3 Motivation and Requirements

Inspired by the success of fuzzing, researchers have adapted fuzzers to test scripting languages execution engines. We first summarize the requirements of scripting language fuzzing, *scalability* and *semantical correctness*. We then discuss how existing research, despite attempts, only partially addressed them, as well as the key limitations that degrade their practicality and completeness.

Supporting multiple languages, implementations and language features. The scalability of existing fuzzing approaches for scripting language execution engines is hampered by three interrelated challenges: the prohibitive expense of developing specialized fuzzers for each language, illustrated by FUZZILLI's extensive lines of code for JavaScript alone; the tedious and error-prone task of adapting fuzzers to the nuanced differences in semantics across alternative implementations of scripting languages, with similar syntax yet often times diverging semantics [5, 9, 19, 20]; and the constant evolution of language features in the standard library, as evidenced by the CPython module index's yearly addition of new global symbols. We investigated the CPython module index from v3.2 (Feb 20, 2011) to v3.12 (Jul 09, 2024)[1] and observed that it adds approximately 493 new global symbols per year (including modules, classes, or attributes), culminating in 15k global symbols as of v3.12. To handle that many symbols, an analyst would have to encode about one symbol *each day*. Any mistakes, such as forgetting to encode a symbol, may lead to incomplete bug discovery.

```
irb> [].first
=> nil # Type: Nil
irb> [1].first
=> 1   # Type: Integer
irb> [1].first(3)
=> [1] # Type: Array
```

(a) Method `Array.first` with different return types.

```
Array.first(num: Integer?)
  => Nil | T | Array<T>
```

(b) Possible return types of `Array.first` at static time.

Figure 2: Motivating example on imprecision of static type specifications. While static analysis might infer that `Array.first` may have one of multiple types, reflection can obtain the *precise* types of the method parameters and return value for a given runtime instance.

Requirement 1: Scalability. An ideal scripting language fuzzer covers multiple scripting languages, implementations as well as language features without requiring significant manual efforts.

Existing approaches fail to scale across multiple languages as well as the language features within a language. Specifically, they require either manual grammar specifications that thoroughly include all language features [21, 50] or large feature-rich initial corpora to infer or recycle language details [37, 46]. Heavy reliance on manual efforts is prone to be incomplete and, worse yet, static approaches call for continuous maintenance of the specifications or corpora as the number of language features grows. Table 3 shows the number of symbols in the manual grammar specifications (NAUTILUS and POLYGLOT), which not only takes a significant portion of grammar specifications but also needs to be updated whenever new language features (e.g., new library APIs) are introduced. This process requires exhaustive manual efforts and hence is not scalable. We argue that manual effort is best spent to extend the coverage beyond what can automatically be covered automatically, e.g., through reflection.

Ensuring the semantical correctness of the mutated programs. Syntactic correctness is easy to achieve through a syntax grammar while semantical correctness remains challenging. This is mainly due to the dynamic type systems of scripting languages. Semantical correctness ensures that a program executes up to a certain point without triggering any exceptions due to missing definitions or other violations. Therefore, with semantical correctness, a scripting language fuzzer can explore more functionalities of the execution engine and reach deeper code.

[1]We counted `<a href="library/...">` for the number of modules in the index page and `<dt ...>` for the number of in-module objects in each module page.

Requirement 2: Semantic Correctness. An ideal scripting language fuzzer generates and mutates semantically correct programs to explore deeper functionalities by avoiding early exceptions.

Existing approaches fall short when generating or mutating *semantically* correct programs. Specifically, manual semantic annotations are prone to under- or over-constraining semantic information, and statically analyzing program semantics is incompatible with the dynamic typing rules in scripting languages. Figure 2 shows the difficulty of statically specifying a type-correct method signature in Ruby. The precise return type of the `Array.first` method may depend on the length of the array (line 2), the elements of the array (line 4), and whether the method is called with an optional parameter (line 6). Static analysis will conclude three types of `Array.first` that has only *one valid type* at runtime, resulting in either producing semantically incorrect programs or overconstraining the testable program space.

4 Reflecta Design

The core novelty of Reflecta is using reflection to fuzz multiple scripting languages with minimal manual intervention. Reflecta only requires a minimal, non-increasing core syntax of the target language and automatically expands it to most language features (e.g., library modules) through reflection. Furthermore, Reflecta leverages reflection to automatically generate type-correct programs (e.g., type-correct calls), achieving high correctness.

Overview. Figure 3 gives an overview of the fuzzing loop. Reflecta first obtains a list of available language features using reflection and generates a small number of initial programs in the corpus with support from the provided core syntax (subsection 4.1). Then, Reflecta probabilistically generates more programs or mutates the corpus programs in a semantic-aware way by deducing type-correct method prototypes with reflection and transplanting necessary objects for new method calls with program slicing (subsection 4.2). Program Generation (subsection 4.1) and Type-Aware Mutation (subsection 4.2) ensure both semantic correctness (generating type correct program) and scalability (operating on the same minimal syntax). Finally, Reflecta takes special care of the programs with unseen object types by adding them to the corpus in the hope of discovering new behaviors of the execution engine following new and rarer types (subsection 4.3). Type-enhanced Corpus Feedback (subsection 4.3) primarily boosts scalability and it makes sure that the fuzzer mutates objects of different types uniformly and scales to all builtin functions and objects without manual scheduling effort.

4.1 Program Generation

Reflecta first discovers available language features in the execution engine through reflection, and then using the minimal core syntax provided by users, generates a small number of initial programs to populate the initial corpus. Reflecta also probabilistically generates new programs during the fuzzing alongside the mutation.

Core syntax and reflection wrappers. Reflecta requires two pieces of non-increasing and one-time manual information: the *core syntax* and the *reflection wrappers*. The core syntax is an invariable language grammar to language feature changes, such as function calls or variable assignments. Each scripting language has its own core syntax and different scripting language execution engines share the same core syntax (see Figure 3). The core syntax is small. Table 3 demonstrates that the core syntax used by Reflecta is 5× to 100× smaller than full-blown syntax used by other fuzzers. Reflecta utilizes the core syntax to generate syntactically correct programs and expands these programs further with semantic information during the mutation phase. The reflection wrappers link Reflecta to the reflection of different target scripting languages (see Table 2). Reflecta utilizes the reflection wrappers to invoke the underlying language-specific reflection APIs and retrieve the data in a JSON format (see Figure 5). Appendix A summarizes how these wrappers are implemented for the evaluated languages.

Wrapper	Return
`enumGlobalSyms()`	Return the list of global symbols (e.g., standard library modules or functions).
`reflectObject(o)`	Given an object `o`, return its type name, attribute list, arity, and callability.

Table 2: Reflection wrappers that Reflecta requires.

Discovering language features. When the fuzzing campaign starts, Reflecta collects available language features by making a reflection call (e.g., `enumGlobalSyms()`) that queries the execution engine for a list of built-in objects, functions, and modules. Next, Reflecta serializes the returned list to a JSON string (see Figure 5) and writes it to a memory-backed file in a fixed location (e.g., `/tmp/fuzzout`). Then, Reflecta deserializes what in the memory-backed file and stores it for later use during mutations. Reflecta does this query only once because globally defined symbols do not change throughout the campaign. At this point, Reflecta only knows about the name of the symbols. Their types (e.g., module or function) are to be determined during the fuzzing campaign.

Generating initial programs. Leveraging the core syntax, Reflecta generates programs containing simple literals (e.g., integers, booleans, null/nil, floats, and strings) and their assignments to variables (e.g., a = 1; b = 1). Reflecta also inject language features (i.e., built-in objects discovered earlier through reflection) into the program by assigning them to new variables. Reflecta then attempts to create more complex values by combining existing literals with unary/binary operators (e.g., `+, -, *, ==, &&`) as part of the core syntax (e.g., c = a + b). More complex operations, such as referencing a property or a method of an object, are delayed to the mutation phase (subsection 4.2) as Reflecta has no semantic knowledge about the generated program at the beginning.

Reflecta deliberately avoids generating control-flow structures due to following three reasons. First, generating control flow structures is unlikely to uncover new bugs, as control-flow logic in interpreters is typically well-tested and simple to implement (e.g., setting virtual machine program counters to another location), a setting that reflection cannot improve upon. The PHP test suite contains over 18,000 test cases, including inputs that crash the interpreter. A recent study [42] reveals that 96.1% of the tests exhibit sequential control flow, executing without branching. This finding suggests that control flow contributes little to the overall code semantics, especially for interpreters. Second, bugs related to

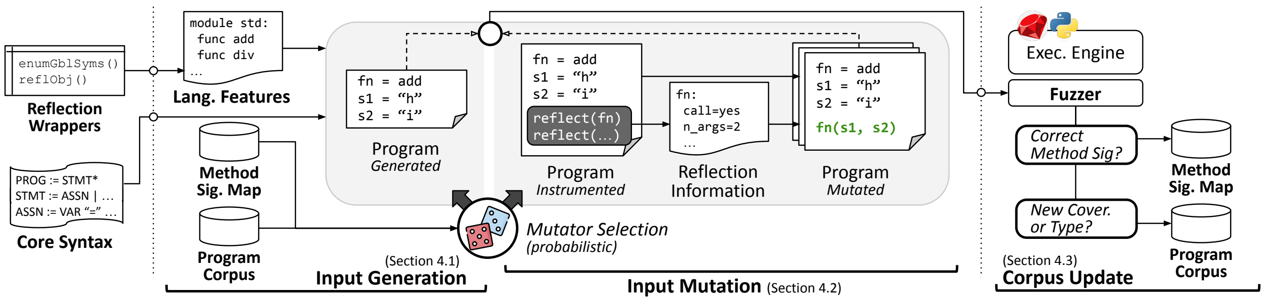


Figure 3: Overview of the REFLECTA fuzzing loop.

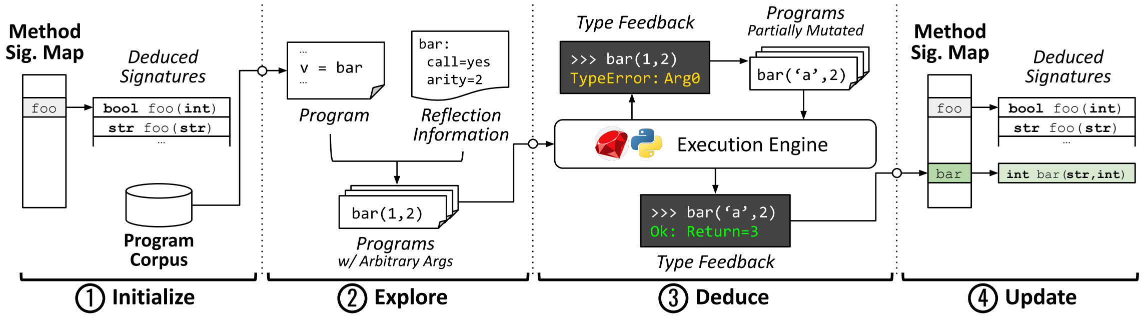


Figure 4: Workflow of method signature deduction (subsection 4.2). The initial map contains foo for explanation purposes.

control flow usually stem from complex JIT optimizations, particularly in browser JavaScript engines, which is already extensively covered in related work [34, 37, 46]. These works focus on specific optimization-related bugs and are largely orthogonal to REFLECTA's approach. REFLECTA, on the other hand focuses on *different targets*, which aims to fuzz a wide range of languages and engines with minimal manual effort. A broader discussion on the language features covered by REFLECTA can be found in section 7.

The generated programs are saved to the corpus if they (1) do not throw exceptions (i.e., syntactically/semantically correct) and (2) find new coverage. In practice, since programs are relatively small (around 10 lines), the percentage of rejected programs is low. Finally, the fuzzer proceeds to the mutation phase when the most recently generated 100 programs do not introduce any new coverage.

On startup, REFLECTA discovers available language features with reflection and generates initial programs with minimal core syntax, injecting the discovered language features to the corpus.

4.2 Type-aware Program Mutation

REFLECTA mutates the corpus programs in a semantic-aware way. Using reflection, REFLECTA first retrieves the semantic information of a selected variable or object attribute. If the selected variable or attribute is not callable, REFLECTA inserts a dereference operation. In case it is callable (i.e., method), REFLECTA takes special care to ensure semantic correctness by deducing its type-correct prototypes and feeding valid arguments to them.

Collecting per-program semantics. To obtain semantic information of a program, REFLECTA tags a program with reflection calls at the end and executes the tagged program to retrieve the reflection information. Specifically, REFLECTA appends a reflection call at the end of the program for each defined variable (i.e., `reflectObject(x)` for the variable x), which in return yields (i) its type name, (ii) its associated attributes, (iii) its arity (i.e., the number of accepted arguments), and (iv) its callability. Figure 5 shows an example of the reflection return. Given a simple program that contains the definition of num (line 1), REFLECTA adds a reflection call at the end of the program to retrieve the reflection information of the variable num (line 2), e.g. type and associated methods.

```
1  num = 123            # Original program.
2  reflectObject(num)   # Added in the mutation phase.
```

(a) Program that reflects on the object num.

```
"num": {
  "type" : "Integer",
  "arity" : "0",
  "methods" : ["anybits?", "nobits?", "downto", ...],
  "properties" : [],
  "callable" : "false",
}
```

(b) Reflection result in JSON.

Figure 5: Example of object reflection.

Selecting variables and attributes. After collecting the semantic information, REFLECTA selects variables or their attributes to add extra operations to them. We note that certain types appear more frequently than others, such as the primitive types like integers and strings. As such, if REFLECTA randomly selects variables from a program, it most likely selects frequent types again and exhausting already covered engine parts. To address this issue, REFLECTA biases the variable selection to the ones with less frequent types. To be specific, Reflecta maintains a global *type count map* that records the number of times a given type appeared in the corpus programs. The probability of selecting a variable is then set inversely proportional to the number of times its type has occurred in the corpus to exercise rare types more frequently.

Reflecta then selects an associated attribute of the selected variable. Reflecta again faces a similar issue with the attribute selection, as certain attributes are defined in the base class and thus appear in the objects more frequently. For example, in Python, the magic methods such as `__class__`, `__sizeof__`, `__init__` are defined for every object, likely randomly choosing those common attributes repeatedly, which results in uninteresting executions. To avoid this, Reflecta maintains a global *attribute count map* that counts the number of times a particular attribute has been selected and selects a new attribute inverse proportionally to the selection count.

Deducing method signatures. If the selected variable or attribute is a method (callable), Reflecta needs to call it with a valid signature so the execution engine can reach the deeper logic without premature exit. Moreover, Reflecta tries to cover *all possible method signatures* for a given method because, due to the dynamic typing rules, methods in scripting languages can accept multiple types as the same argument. Calling a method with different signatures may lead to other interesting behaviors, as depicted in Figure 2a.

To address this, Reflecta automatically discovers valid method signatures using reflection and type error feedback, keeping valid signatures in the *method signature map*. Figure 4 describes the workflow of method signature deduction. (1) At the beginning of a fuzzing campaign, Reflecta first initializes an empty method signature map that maps a method name to a set of valid signatures. A valid signature consists of a list of argument types and a return type. (2) Reflecta then explores valid signatures during fuzzing by leveraging the arity (i.e., the number of arguments) information of a given method, producing method calls with the correct number of arguments with various combinations of argument types. (3) Reflecta incrementally deduces a set of correct method signatures (with optional parameters) using a type error thrown by the execution engine that suggests which argument has a wrong type. (4) Finally, if the produced method call does not throw any type error, Reflecta instruments all method calls in the mutated program with reflection, extracts the type of their arguments and return values, and adds new method signatures to the map.

Exploring and exploiting method signatures. Leveraging the method signature map, Reflecta then utilizes an $\epsilon-greedy$ method to balance the signature exploration and exploitation. As the fuzzing campaign progresses, the method signature map is populated with more signatures. When Reflecta selects a method as a mutation target, Reflecta tests whether a randomly chosen value δ in [0, 1] is smaller than ϵ to decide to explore more method signatures with arbitrary arguments (*exploration*), otherwise utilizes a type-correct signature from the method signature map (*exploitation*). We tested ϵ from a wide range of [0.01, 0.5] and empirically selected 0.2 as our final choice because it worked best for all targets in our development and experimentation.

Adding correct method calls. When adding a new method call, the correct arguments may not exist in the target program. Moreover, other than primitive types such as integers and strings, Reflecta has no prior knowledge of properly constructing a certain type. To circumvent this, Reflecta leverages existing instantiation of such types in the corpus. This is especially effective for Reflecta, as the valid signature entry in the method signature map guarantees that there must be *some* corpus programs that have properly instantiated all the types in the signature learned previously. Leveraging this, Reflecta extracts a variable with the desired type from such programs by backward slicing the variable and its dependencies. Specifically, starting from the statement that defines the desired variable, Reflecta recursively marks all dependent statements that are referenced by the statement and its recursively dependent statements. Reflecta then transplants the variable and its program slice to the target program before the new call.

Reflecta uses reflection to collect semantic information (type, attributes, arity, callability) before mutation. It randomly selects target variables and attributes, favoring rare types via a count map, and balances exploration and exploitation of method signatures using an ϵ-greedy approach. Type-correct method calls are created by transplanting valid type instantiations from the corpus.

4.3 Type-enhanced Corpus Feedback

Overview. Reflecta inspects the *object type* of every variable in a program with reflection and saves the program in the corpus if some variables have unseen types, considering that different object types permit other program behaviors. Specifically, an object type defines which operations are implemented on the object and which methods it can be passed to as an argument. They directly indicate new associated program behaviors, similar to the conventional coverage feedback indicating potentially new code areas in general-purpose fuzzing [7, 32, 48]. Reflecta utilizes both for corpus feedback.

New type without coverage gain. Notice that new types in the program do not necessarily result in new code coverage, yet can trigger new program behavior. For example, if an object of a certain type has been implicitly instantiated by the execution engine (e.g., during the engine initialization) or by other language features (e.g., standard library), newly instantiating such types in the mutated programs would not result in new coverage as the prior implicit instantiation already covered them. However, since the implicit instantiation is out of reach for Reflecta, the observable program behaviors from such types are highly limited. Instead, by having a type instance in the program, Reflecta can better explore its associated program behaviors by trying values and arguments of various types. While similar, the global type count map (subsection 4.3) and the global attribute map (subsection 4.2) serve different purposes. The global type count map counts how many unique types the fuzzer has generated so far. The global attribute count map keeps track of the attributes for each unique type. The two maps are used to ensure uniform generation of unique types, and their attributes. Without the maps the fuzzer tends to be biased towards specific types, or specific attributes of a certain object type.

Handling meta-types. Since scripting languages also follow the object-oriented paradigm, the class and module definitions themselves are also objects, hence having their own types. Specifically, all class definitions have the equal meta-type (e.g., `Class`) regardless of their definition, and so do modules (e.g., `Module`). This potentially disrupts the type-based corpus feedback as some classes and modules are *statically* referenced without instantiation, where reflection on such references would yield the same meta-type. To differentiate the meta-types by their specific definitions (e.g., class for `Integer` or `String`), Reflecta appends their definition names to the meta-type name (e.g., `Class#Integer` and `Class#String`). This way, when a variable that references a new class or module definition appears, Reflecta can distinguish different definitions of the same meta-type and add the program to the corpus.

Reflecta uses new types, alongside code coverage, as a criterion for corpus addition. This type-enhanced feedback allows Reflecta to explore language features (e.g., standard library objects) within generated programs. It also disambiguates object meta-types by associating them with their definition names (e.g., `Class#Integer` for `Integer`).

5 Implementation

We base our implementation on Fuzzilli, a mature fuzzer maintained by Google for the Chrome browser's V8 JavaScript engine. Fuzzilli utilizes intermediate representation (IR) for semantic mutation and an in-memory representation of common language constructs, for example, an assignment statement, or a binary operation. Mutators and Generators that create unsupported IR statements are removed to prevent introducing such constructs into the program. We reuse the basic program generation and mutation facilities of Fuzzilli and extend it with language feature discovery (subsection 4.1), the type-aware program mutation (subsection 4.2) as additional mutators, and the type-enhanced corpus feedback (subsection 4.3).

Core syntax and reflection wrapper. For the core syntax of each scripting language, we implement a lightweight Fuzzilli lifter that is responsible for serializing IR into source code strings that the execution engine accepts. The lifter has 90 lines of Swift code on average across 4 different languages. Notice that implementing lifters is only a one-time effort, as they only specify the core grammar and do not contain any language feature details. For reflection wrappers, we add 80 lines of code on average in each language. We provide more implementation details in Appendix G and Appendix A.

6 Evaluation

In this section, we compare Reflecta to state-of-the-art scripting language fuzzers to answer the following:

RQ1 How do the programs produced by Reflecta compare to hand-tuned fuzzers in semantic correctness?
RQ2 How much can Reflecta cover the language features with minimal manual effort?
RQ3 Can Reflecta find new bugs in real-world scripting language execution engines?
RQ4 How challenging is adding new targets with Reflecta?

Language		Fuzzer				
		Nautilus	PolyGlot	Fuzzilli	PyRTFuzz	Reflecta
JavaScript	Grammar	972	469	881	-	91
	Symbol	852	71	*1,181	-	0
PHP	Grammar	8,719	647	-	-	86
	Symbol	8,655	2,598	-	-	0
Ruby	Grammar	1,176	388	-	-	87
	Symbol	1,142	0	-	-	0
Python	Grammar	637	615	-	-	83
	Symbol	412	0	-	*1205	0

Table 3: Grammar specification for evaluation used in each fuzzer. *: counting includes both identifier names and semantic annotations (e.g., method signatures).

6.1 Experimental Setup

Targets languages. We choose Python, JavaScript, PHP, and Ruby as they ranked as the top-four most popular scripting languages according to the Stack Overflow 2023 Developer Survey [17]. For Python and Ruby, we choose two alternative language implementations to demonstrate the self-adaptive capability of Reflecta for different execution engines, even within a single language. All targets are compiled with AddressSanitizer [49] except for Ruby due to incompatibility (ASAN executable crashes). We reuse existing or create persistent drivers for all targets to improve the performance, except for Nautilus that utilizes its own incompatible forkserver. The edge coverage results are obtained off-line using a separately built binary with SanitizerCoverage [16] instrumentation.

Baseline fuzzers. We choose three state-of-the-art scripting language fuzzers as our baseline: Nautilus, PolyGlot, and Fuzzilli. Unfortunately, we could not compare against SoFi as it was not available as open source (as noted by related work [33, 34]). We also compare to PyRTFuzz, as its dynamic refinement technique is similar to Reflecta's use of reflection. We note, however, PyRTFuzz is highly specialized towards CPython, requiring the `sys.settrace` function, which requires a custom MicroPython build and is not available for languages like JavaScript. We attempted to reproduce PyRTFuzz's results, but ran into compatibility issues when generating API specifications for Python 3.12. We present a best effort replication and comparison against PyRTFuzz leveraging bugs reported in their paper and their replication package on an older Python 3.9 in Appendix D and Appendix E.

For Nautilus, we reuse the grammar provided in their repository for PHP, JavaScript, and Ruby and adapt the Python grammar from the ANTLR grammar repository [3] following their documentation. Appendix B explains how we adapted the Python grammar. For PolyGlot, we again adapt the grammar from the ANTLR repository following the original paper and reuse the semantic annotation for PHP provided in their artifact repository (since the semantic annotation support of PolyGlot for other languages remains under development as of this writing.) By default, we supply PolyGlot with the initial programs created by Reflecta as the initial corpus (following [21, 24]). We conduct a separate evaluation for PHP for which PolyGlot provides an initial corpus (PolyGlot+C).

Grammar specification statistics. Table 3 shows the statistics of grammar specifications used in the evaluation, where the "Grammar" and "Symbol" specify the number of grammar rules and symbols (e.g., standard library methods) in the initial syntax. For Nautilus and PolyGlot, we count the number of grammar rules and symbols in the grammar specifications. For Fuzzilli and Reflecta, we count the LoC of grammar statements and specified identifiers in the IR lifters. We additionally include the LoC of semantic annotations of Fuzzilli and PyRTFuzz as "Symbol". We provide snippets of grammar used by Reflecta, Nautilus, and PolyGlot, and links to their full grammar in Appendix G. PyRTFuzz requires static analysis of CPython's APIs. Moreover, inference of runtime types deviates from the description in the paper, with the artifact suggesting potential need for manual specification [12]. Note that PyRTFuzz does dynamic type refinement in CPython only once before fuzzing, so polymorphic functions will be still tested with a single static type signature (monomorphic). Reflecta, however, does runtime reflection and type inference for polymorphic functions, allowing for broader and deeper exploration. Importantly, Reflecta starts with significantly smaller grammar than any other fuzzers without any semantic information. We demonstrate how to integrate other targets into Reflecta in Appendix C.

Evaluation configuration. We run all experiments on a cluster node of a 16-core Intel Xeon 5218 CPU machines with 64GB of memory (with hyper-threads disabled). Each fuzzing instance is pinned to a single core using `cpuset`. The comparative experiments are 24 hours and repeated eight times for each fuzzer-target pair. The bug-finding fuzzing campaign is performed over an extended span of three months. All bugs were found within 24 hours.

6.2 Semantic Correctness

To answer **RQ1**, we calculate the aggregate semantic correctness rate of the produced programs similarly to other fuzzing papers [29, 34, 37, 46]. For Reflecta and Fuzzilli, we divide the number of correct executions (with the exit code zero) by the number of total executions (accurate statistics recorded during fuzzing). For PolyGlot and Nautilus, we replay the saved corpus after a 24-hour campaign to calculate the correctness rate, similarly. For a fair comparison, we exclude the duplicate minimized copies in the Nautilus corpus, since the minimized copy and the original saved input have the same coverage. Table 4 shows the correctness rate of each fuzzer on different targets. Except for Fuzzilli on V8 that utilizes heavily hand-tuned manual semantic annotations, Reflecta achieves the highest correctness of 62.3%, while Nautilus and PolyGlot only achieve 35.7% and 20.2%.

	Nautilus		PolyGlot		Fuzzilli	Reflecta
Ruby	34.8%	1116/3207	19.8%	421/2127	-	61.0%
MRuby	34.7%	821/2364	25.7%	397/1544	-	62.5%
CPython	28.3%	711/2513	14.0%	158/1127	-	53.1%
MicroPython	36.6%	1430/3907	22.3%	131/587	-	62.1%
PHP	40.1%	1966/4903	29.7%	1323/4455	-	70.8%
V8	39.3%	419/1066	9.8%	291/2973	67.7%	64.2%
Average	35.7%		20.2%		67.7%	62.3%

Table 4: Semantic correctness rate and Correct/Total ratio of produced programs. -: Fuzzilli only supports JavaScript.

We manually inspected the corpus programs from Nautilus and identified the following recurring reasons for its low correctness: (1) dereference error, dereferencing a statically-provided list of attributes that do not belong to an object and (2) type errors, calling a method with wrongly-typed arguments. In contrast, Reflecta does not suffer from such problems as it obtains precise semantic information from reflection. In particular, Reflecta obtains the valid attributes of a given object at runtime through reflection and automatically deduces and exploits type-correct method signatures using the $\epsilon - greedy$ algorithm. PolyGlot reports to support language semantics through a feature called semantic validation. However, the feature is still under development and not supported as of writing this paper. Moreover, the feature is not meant for dynamically typed languages such as most scripting languages, where PolyGlot uses `AnyType` for any object types undecidable at static time. This suggests that the feature would have caused semantic incorrectness even if it were enabled

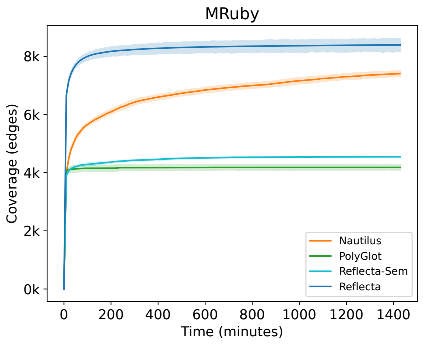
Incorrect programs produced by Reflecta are mainly due to semantic requirements beyond type correctness. For example, methods and functions that require stateful invocation, ordering, and arguments that expect a specific format of strings (e.g., xml). Such requirements go beyond the capability of reflection and require additional logic or manual intervention. Reflecta reports a relatively lower correctness rate in CPython compared to other targets. This is because of CPython's relatively large standard library that slows down method signature deduction to explore new signatures before exploiting them. Moreover, Python is considered "more strongly-typed" than Ruby, PHP, or JavaScript where implicit type casts and coercion is common. This may also complicate type-correct signature deduction and cause relatively more type-incorrect executions.

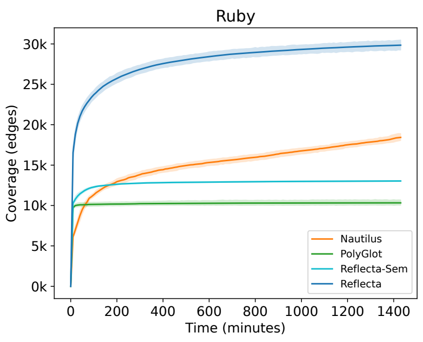
Reflecta does not require prior knowledge of semantics and achieves superior correctness compared to the state-of-the-art language-general scripting language fuzzers. Reflecta is comparable to specialized fuzzers with extensive manual semantic annotations.

6.3 Code Coverage

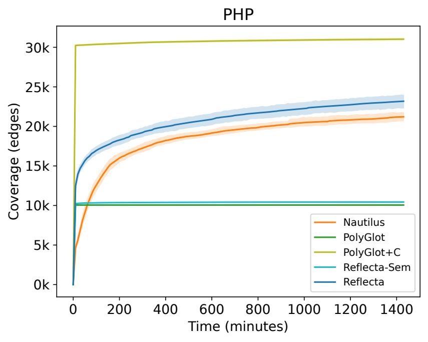
To answer **RQ2**, we measure the edge coverage of each target fuzzed by each fuzzer. Figure 6 shows the coverage growth over the 24 hours of fuzzing campaigns on various targets, where the shades indicate a 95% confidence interval.

Languages with different implementations. Figure 6a and Figure 6b show the coverage growth of the fuzzers on two different language implementations of the same language (Ruby vs. MRuby and CPython vs. MicroPython), where Reflecta outperforms Nautilus in reference implementations (Ruby and CPython) even more than in the alternatives (MRuby and MicroPython). This is because the reference implementations have larger standard libraries than the alternatives, some of which are missed by the pre-engineered grammar by Nautilus. Reflecta, on the other hand, automatically *scales* the minimal syntax to all libraries and adapts itself to the underlying execution engines. Reflecta-Sem shows the ablated Reflecta without type-correct mutations, showing that the *semantic correctness* support significantly helps Reflecta reach deeper code.

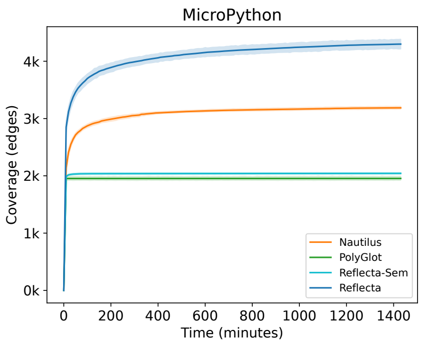


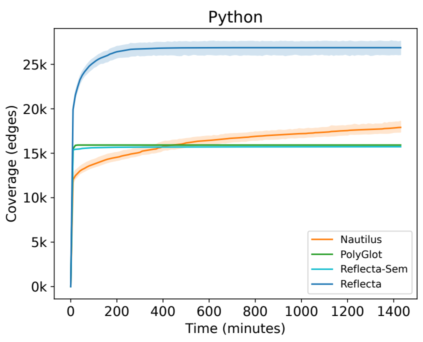


(a) Ruby

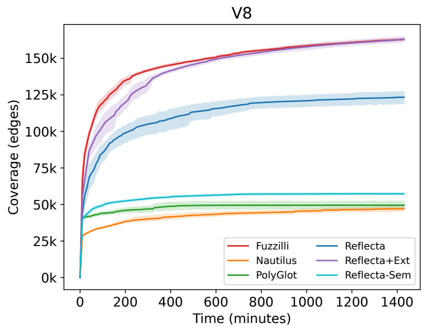


(c) PHP





(b) Python



(d) JavaScript

Figure 6: Edge coverage growth over 24 hours on different targets with 95% confidence intervals.

Target	Globals	Attributes	Methods	Explorations Per Fn.
Ruby	81	708	1,339	152
MRuby	67	453	808	486
CPython	63	2,349	2,377	129
MicroPython	20	102	293	6182
PHP	772	0	0	878
V8	48	353	304	170

Table 5: Median language features learned and explorations per function by REFLECTA across eight runs.

PolyGlot stopped discovering new coverage after few minutes in every campaign, while we double-checked that all campaigns terminated normally after 24 hours. Two relevant factors are: (1) PolyGlot highly depends on the initial corpus for language features (e.g., standard library symbols), which is also shown by the high coverage gain in PHP (PolyGlot+C in Figure 6c) that utilizes the initial corpus given by PolyGlot, in contrast to the other targets with our generated initial corpus. (2) As noted in subsection 6.2, the semantic validation feature of PolyGlot is incomplete in the publicly-available version, degrading the exploration capability significantly. This is supported by a similar coverage trend in Reflecta-Sem, where Reflecta's semantic correctness features are disabled.

Figure 8 shows the coverage differential between Reflecta and Nautilus on Ruby, where green and red indicate Reflecta and Nautilus covering more edges than the other, respectively. An interesting and yet surprising finding is that the implementation

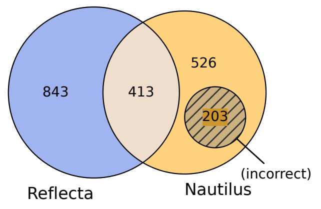


Figure 7: Venn diagram of identifier names in MRuby learned by Reflecta or specified by Nautilus.

of builtin functions and objects in the standard library make up for a significant amount of scripting engine code. More perceived and frequently used language features like control flow and class definition/inheritance, have a relatively simple implementation (evidenced by relatively small size of `vm.c` and `class.c` in Figure 8), and are often better tested due to exposure to more users. Overall, Reflecta covers more edges than Nautilus in most source code files, which is consistent with the coverage plot in Figure 6. Especially, Reflecta outperforms the most in standard library sources, such as `complex.c`, `numeric.{c,h}`, and `generator.c`, demonstrating how thoroughly Reflecta covers built-in methods with reflection.

Languages with intensive manual support. Figure 6c and Figure 6d show the coverage growth on the languages with intensive manual supports for semantics by PolyGlot and Fuzzilli. For both targets, Reflecta achieves comparable code coverage to the best

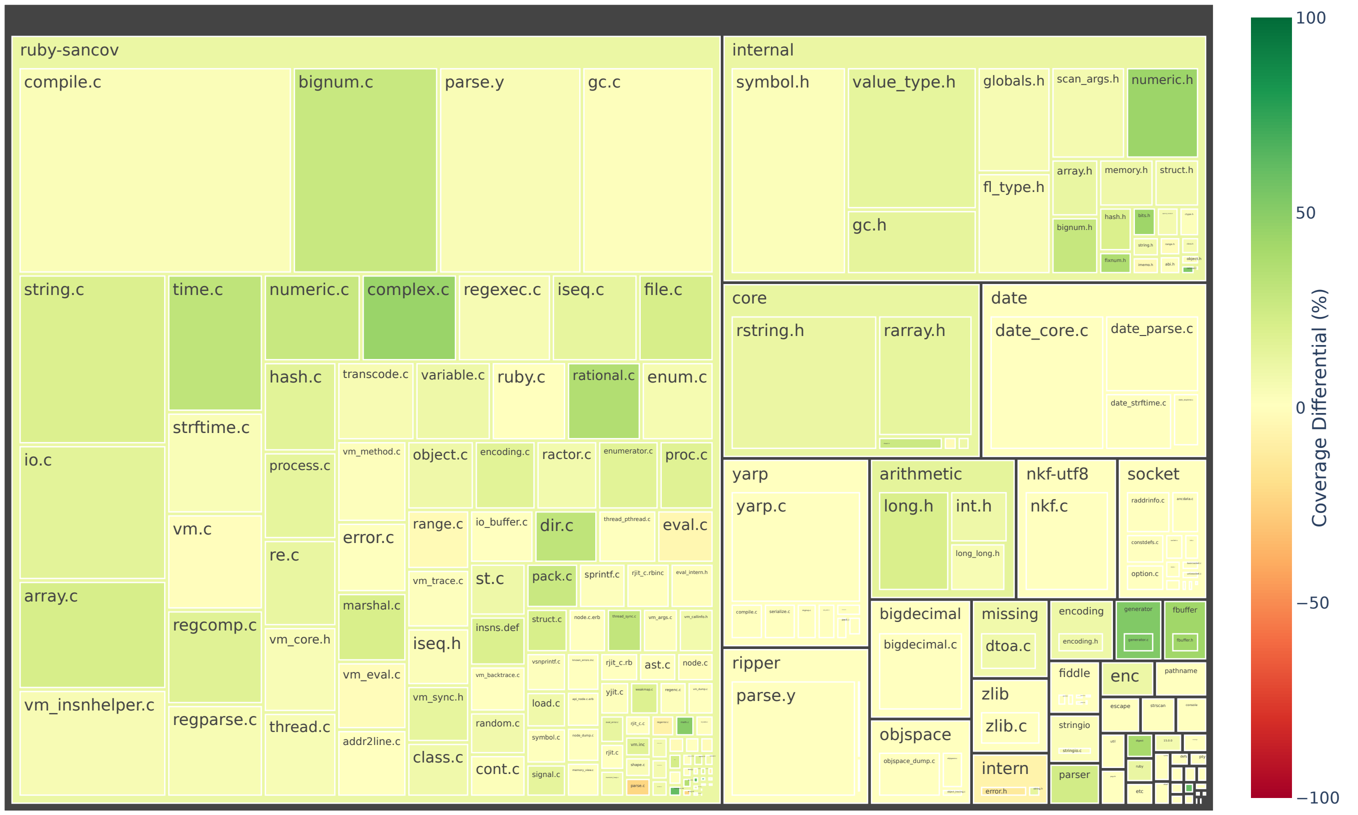


Figure 8: Coverage differential between Reflecta and Nautilus on Ruby. Top-level boxes: directories. Inner boxes: source code files. Box size: the number of edges in the corresponding directory or file. Color: the coverage differential. The coverage differential was calculated by (Edges(Reflecta) - Edges(Nautilus)) / Edges(Total)

configurations of PolyGlot and Fuzzilli even without any manual semantic hints, and continues to expand the code coverage further over time. PolyGlot, on the other hand, gains its coverage predominantly from manual efforts (i.e., the initial corpus) and quickly hits the coverage saturation point. Fuzzilli gains more coverage than Reflecta on V8, and it turns out that the specialized JIT support in Fuzzilli facilitates the JIT logic triggering.

Extended grammar through reflection. Table 5 shows the number of global objects, attributes, and distinct method signatures that Reflecta learned through reflection during the campaign. Reflecta learned no attributes and methods in PHP as most standard library methods in PHP are global; PHP was historically a procedural language and adopted the object-oriented paradigm later. For example, popping an element from an array from in PHP uses the `array_pop(arr)` global function, whereas in JavaScript, Ruby and Python it is expressed as `arr.pop()`. This explains the relatively small margin between Reflecta and Nautilus on PHP.

Recall that Reflecta employs an ϵ-greedy approach to continuously explore learned method signatures. This means Reflecta never definitively concludes that a given method or function is fully explored, but rather continues to invoke it with new argument combinations throughout the entire campaign. We modified Reflecta so that when it attempts to *explore* the signature of a function or method—i.e., trying to invoke it with arguments of previously unseen types—it records these attempts in a global counter. The last column of Table 5 shows the average number of times a method/function is explored over a 24-hour period. We also dump the learned signature map after each campaign and find that, for the majority of functions/methods discovered (97.3%), fewer than six polymorphic signatures are learned (fewer polymorphic signatures requires fewer exploration trials). Furthermore, each function or method is exercised at least 150 times throughout the fuzzing process, ensuring extensive exploration of its behavior.

Figure 7 provides a comparative analysis of the language features learned by Reflecta and specified by Nautilus for MRuby. Reflecta learned a total of 67 globals, 453 attributes, and 808 methods, the union of which sums up to 1,256 identifier names (some methods and attributes have the same name). In comparison, the grammar specification of Nautilus for MRuby does not distinguish between global objects and methods and has a total 1,142 production rules for identifier names.

Interestingly, despite the total number of identifier names learned by Reflecta and specified by Nautilus being similar, the intersection is only 413 (32.9% of all Reflecta learned identifiers). We further investigated the MRuby grammar specification for Nautilus and found out Nautilus includes some ambiguous identifiers in the grammar specification. For example, it contains identifiers such as `kh_del_{ht,iv,mt}` that do not appear in the entire MRuby source repository. In total, we discover 27.8% of the unique identifiers in Nautilus's MRuby grammar are incorrectly specified. We assume they are artifacts of scraping the built-in names from the documentation. Incorrect specifications lead to wasted execution cycles in Nautilus, which explains the lower coverage gain as well as a slower coverage growth rate.

With only the minimal core language syntax, Reflecta achieves significantly better code coverage compared to SoTA language-general scripting language fuzzers, as well as is comparable to language-specific fuzzers with intensive manual semantic supports.

6.4 Discovered Bugs

To answer **RQ3**, we continuously ran Reflecta and other baseline fuzzers over an extended period of three months during development. Table 6 shows all bugs discovered by Reflecta. Reflecta found a total of 25 unique bugs, including 8 in MRuby, 14 in MicroPython, and 3 in PHP. Notice that MRuby and PHP have been extensively fuzzed by both prior work [21, 32, 50] and the OSS-Fuzz [10] continuous fuzzing platform. This showcases the deep bug finding capability of Reflecta in diverse language execution engines. Of the baseline fuzzers, only Nautilus discovered three new bugs in MicroPython with the new manually specified grammar provided by us. It's noteworthy all bugs found by Nautilus is a subset of the bugs found by Reflecta. Reflecta did not find new bugs in V8. Mentioned in the section 4, Reflecta does not generate any control flow structures, as they are orthogonal to the semantic information provided by reflection. Most recent bugs in V8 are in the JIT optimization pipeline [46]. This requires loops to make code hot and trigger JIT optimization. Reflecta primarily finds bugs in standard libraries, and allows for straight-forward fuzzing of other scripting languages. We discuss this limitation in the discussion section section 7. The following case studies (with three additional case studies in Appendix F) explain how Reflecta found them and why existing fuzzers missed them.

Case study: stack-overflow at mrb_vm_exec in MRuby. Figure 9 shows the minimized proof-of-concept program for the stack overflow bug at `mrb_vm_exec` in MRuby. The program first creates a lazy iterator over an empty dictionary (line 2) and makes a circular reference to itself using `args=` and `filter_map` (lines 5-6). Finally, it triggers an iteration by calling the `count` method, causing a stack overflow due to the infinite internal dereference (line 9).

Nautilus failed to find this bug as its grammar specification did not include the `filter_map` and `args=` methods, showing that a human-written grammar is prone to under-specification; missing method names can lead to missing potential bugs as shown here. PolyGlot similarly missed this bug as the initial corpus did not contain such methods. On the other hand, Reflecta discovered this bug thanks to two major reasons. First, Reflecta automatically recognized `args=` and `filter_map` through reflection, which made testing such methods possible to begin with. Second, Reflecta learned the correct method signature for `args=` that accepts one iterator type, which allowed Reflecta to quickly test the method with correct semantics. The type-enhanced corpus feedback also

Target	Bug ID	Bug Type	Crash Location	Status
MRuby	6051	null-deref	mrb_addrinfo_unix_path	Fixed
	6052	null-deref	mrb_vm_exec	Fixed
	6065	null-deref	eval_under	Fixed
	6066	null-deref	mrb_struct_to_h	Fixed
	-	heap-uaf	ary_rotate_bang	Fixed
	-	heap-uaf	ary_compact_bang	Fixed
	6067	null-deref	mrb_string_value_cstr	Fixed
	6068	stack-over	mrb_vm_exec	Confirmed
MicroPython	12522	null-deref	mp_reader_new_file	Fixed
	12528	heap-buf	mpz_as_bytes	Confirmed
	12532	global-buf	mp_get_stream_raise	Confirmed
	12543	heap-uaf	__bt_get	Fixed
	12562	null-deref	mvlsb_fill_rect	Fixed
	12587	global-buf	mp_vfs_umount	Confirmed
	12605	heap-buf	mp_seq_multiply	Confirmed
	12605	null-deref	mp_obj_equal_not_equal	Confirmed
	12660	heap-buf	mpz_hash	Fixed
	12670	segfault	vfs_posix_file_ioctl	Confirmed
	12702	global-buf	uctypes_struct_agg_size	Fixed
	12735	segfault	decompress_error_text_maybe	Confirmed
	12776	global-buf	task_push_queue	Confirmed
	12830	global-buf	mp_obj_class_lookup	Fixed
PHP	-	null-deref	zend_hash_clean	Fixed
	-	null-deref	zif_forward_static_call	Fixed
	-	null-deref	zif_func_num_args	Fixed

Table 6: Unique bugs found by Reflecta. Bug ID: issue number. -: Reported through GitHub Security Advisories

```
1  # Step 1: Create a lazy iterator over an empty dict.
2  it = {}.lazy()
3  # Step 2: Create a circular reference.
4  it.args=(it)
5  fmap = it.filter_map()
6  # Step 3: Use the iterator, causing a stack overflow.
7  fmap.count()
```

Figure 9: Minimized proof-of-concept program for the stack overflow bug at `mrb_vm_exec` in MRuby.

enabled faster discovery of this bug as the iterator type occurs less frequently than common literals like strings and integers.

Thanks to automatic scalability and high semantic correctness, Reflecta found 25 unique bugs in MRuby, MicroPython, and PHP, which have not been found by any existing fuzzers or OSSFuzz.

7 Limitation and Discussion

Covered language features. Reflecta does not generate explicit control flow structure (e.g., loops, conditionals, or functions) which would often lead to dead code, resulting in wasted fuzzing cycles and lower coverage. However, not generating control flow structures entirely prevents Reflecta from finding certain bugs like JIT optimization bugs (subsection 6.4). Several approaches have been proposed to address this issue: (1) avoiding control flow constructs altogether by excluding them from the grammar specification [21, 50]; (2) carefully generating control flow conditions and verifying their validity statically [34, 43]; or (3) relying on predefined templates or seed corpora to guide fuzzing [37, 51, 59]. Reflecta complements these mechanisms by providing a strong foundation of semantic correctness, instead of directly mitigating the challenges of generating control flow structures.

Regarding more specific language features, such as prototypical inheritance in JavaScript, these constructs may still require additional manual effort. Nonetheless, since Reflecta provides a solid base fuzzer across multiple scripting languages, it can be extended to handle more features with further refinement.

Broader application of reflection. The primary threat to our evaluation concerns the selection of fuzzing targets and the generalizability of our findings. To mitigate this risk, we chose the top four popular scripting languages, including four standard and two alternative implementations. We demonstrate how much effort is necessary to support a new execution engine (i.e., SpiderMonkey in Appendix C) and also verified that less popular scripting languages (e.g., Lua, Perl, and R) support reflection and can similarly benefit from Reflecta as-is with only minor engineering costs (i.e., providing minimal syntax and reflection wrappers).

Besides scalability and semantic correctness, reflection can also improve other aspects of scripting language fuzzing. For efficiency, reflection helps in corpus scheduling [25, 44, 57] by prioritizing programs with rare types or attributes. Spending more energy on interesting programs may allow fuzzers to explore the program space faster. For testing completeness, reflection can explore the dynamic side of scripting languages, such as deleting object attributes or redefining methods at runtime. This may reveal invalid assumptions of execution engines about the shape of objects, which have historically caused many vulnerabilities in JavaScript engines [35].

8 Related Work

Language-specific fuzzing. To support a language-specific grammar, early research mainly focused on a targeted language. CSmith [55] generates C programs without undefined behaviors using manual expert knowledge on the C grammar. Inspired by CSmith, the EMI class of fuzzers [43, 51, 59], find bugs in compilers by introducing semantic preserving mutations, also known as metamorphic testing [27]. PyRTFuzz [45] fuzzes Python runtime libraries through the tracing module and leveraging the CPython unit test suite. As an orthogonal testing oracle aspect, language fuzzers [23] might also leverage differential testing (by comparing different implementations and configurations) to detect correctness bugs, in addition to finding crashes and memory corruptions. Specialized JavaScript fuzzers [37, 38, 46] adopt a similar approach to CSmith. The language-specific approach, while allowing for more manual control and covering specific language features more deeply, unfortunately, does not scale to multiple languages. Reflecta, on the other hand, provide a good starting point when testing new languages and implementations.

Language-general fuzzing. Some language-general fuzzers [21, 47, 50, 53] are compatible with multiple languages. Nautilus combined grammar with coverage feedback to improve program generation and mutation. Gramatron constructs grammar automatons and performs more aggressive mutations to trigger complex bugs faster. These proposals still suffer from the language scalability issue as they still require full-blown manual grammar specifications with constant updates. PolyGlot supports semantic-aware program mutation for general languages, but it was not designed for *dynamic* languages (i.e., scripting languages), hence is prone to produce incorrect programs.

Syntax inference. Automatic specification inference for language fuzzing mostly focuses on generating syntactically correct inputs. Grimoire [24] uses input generalization to automatically synthesize the structure of inputs. Glade [22] is an iterative algorithm to automatically infer a context-free grammar. Autogram [39], on the other hand, uses taint analysis for grammar inference. However, they only enforce the *syntactic* correctness of the produced programs. Reflecta, on the other hand, enforces *semantic* information from the target language, generating semantically correct programs that are more likely to trigger bugs.

Semantics inference. Specification inference is studied at a more semantic level outside language fuzzing. In OS kernel fuzzing, researchers have attempted to auto-generate system call specifications by applying static and dynamic analysis techniques [28, 30, 36, 52]. In library fuzzing, researchers attempted to infer the correct library call sequence by analyzing library code and its applications [26, 40, 41, 58]. Existing work on OS kernel and library fuzzing typically assumes static typing (C/C++) and have relatively fixed behavior patterns (e.g., memory and resources must be allocated before use and freed only once). In contrast, scripting languages, with their dynamic typing and more fluid execution models, do not conform to such rigid structures, and thus such techniques cannot be applied to scripting language fuzzing.

9 Conclusion

Reflecta introduces a *scalable* and *semantic-aware* approach for fuzzing scripting languages. We achieve this generality through runtime reflection instrumentation, which enables type-aware mutation. Our evaluation shows that Reflecta outperforms state-of-the-art language-general scripting language fuzzers in terms of code coverage and semantic correctness, and is even comparable to heavily hand-tuned language-specific fuzzers. During our evaluation, Reflecta also discovered 25 bugs in popular scripting language engines, showcasing its real-world effectiveness.

Acknowledgments

We thank Florian Hofhammer, Nicolas Badoux, the rest of the HexHive, and the anonymous reviewers for their detailed feedback. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2 186974, a gift from Intel Corporation, and funding from BMK, BMAW, and the State of Upper Austria in the frame of the COMET Module DEPS (grant no. 888338) managed by FFG.

References

[1] [n. d.]. Heap hardening Issue 14083 php/php-src — github.com. https://github.com/php/php-src/issues/14083. [Accessed 19-12-2024].
[2] 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.
[3] 2023. Antlr/Grammars-v4: Grammars Written for ANTLR v4; Expectation That the Grammars Are Free of Actions. https://github.com/antlr/grammars-v4.
[4] 2023. ECMA-262.
[5] 2023. HHVM. http://hhvm.com/.
[6] 2023. LCOV - Unix_coverage_v1.21.0-199-Gcc8fc450a.Info. https://micropython.org/resources/code-coverage/.
[7] 2023. libFuzzer – a Library for Coverage-Guided Fuzz Testing. — LLVM 17.0.0git Documentation. https://llvm.org/docs/LibFuzzer.html.
[8] 2023. Memory Error in Calendar.Isleap · Issue #103687 · Python/Cpython. https://github.com/python/cpython/issues/103687.
[9] 2023. Mruby. https://mruby.org/.

[10] 2023. OSS-Fuzz: Continuous Fuzzing for Open Source Software. Google.
[11] 2023. PHP: Hypertext Preprocessor. https://www.php.net/index.php.
[12] 2023. 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".
[13] 2023. RecursionError in Pyclbr.Readmodule_ex · Issue #103864 · Python/Cpython. https://github.com/python/cpython/issues/103864.
[14] 2023. Reflective Programming. *Wikipedia* (Jan. 2023).
[15] 2023. Ruby Programming Language. https://www.ruby-lang.org/en/.
[16] 2023. SanitizerCoverage — Clang 18.0.0git Documentation. https://clang.llvm.org/docs/SanitizerCoverage.html.
[17] 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023.
[18] 2023. Welcome to Python.Org. https://www.python.org/.
[19] 2024. MicroPython - Python for Microcontrollers. http://micropython.org/.
[20] 2024. Nginx/Njs. nginx.
[21] 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 (NDSS'19)*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2019.23412
[22] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, Barcelona Spain, 95–110. https://doi.org/10.1145/3062341.3062349
[23] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*. ACM, Los Angeles CA USA, 351–364. https://doi.org/10.1145/3548606.3560624
[24] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure While Fuzzing. In *28th USENIX Security Symposium (Security'19)*. 1985–2002.
[25] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (May 2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841
[26] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. HOPPER: Interpretative Fuzzing for Libraries. https://doi.org/10.48550/arXiv.2309.03496 arXiv:2309.03496 [cs]
[27] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. https://doi.org/10.48550/arXiv.2002.12543 arXiv:2002.12543 [cs]
[28] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. Association for Computing Machinery, New York, NY, USA, 749–763. https://doi.org/10.1145/3460120.3484564
[29] 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'21)*. 642–658. https://doi.org/10.1109/SP40001.2021.00071
[30] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3133956.3134069
[31] Author eyalitkin. 2017. MRuby VM Escape – Step by Step.
[32] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT'20)*.
[33] fuzz-evaluator. 2023. Artifact Evaluation: SoFi.
[34] 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 (NDSS'23)*. The Internet Society.
[35] Choongwoo Han. 2023. Case Study of JavaScript Engine Vulnerabilities.
[36] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. Association for Computing Machinery, New York, NY, USA, 2345–2358. https://doi.org/10.1145/3133956.3134103
[37] 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 (NDSS'19)*. The Internet Society.
[38] 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 (CCS'21)*. ACM, Virtual Event Republic of Korea, 2229–2242. https://doi.org/10.1145/3460120.3484823
[39] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. 720–725.
[40] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (Security'20)*. 2271–2287.
[41] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. UTopia: Automatic Generation of Fuzz Driver Using Unit Tests. In *2023 IEEE Symposium on Security and Privacy (SP'23)*. 2676–2692. https://doi.org/10.1109/SP46215.2023.10179394
[42] Yuancheng Jiang, Chuqi Zhang, Bonan Ruan, Jiahao Liu, Manuel Rigger, Roland Yap, and Zhenkai Liang. 2024. Fuzzing the PHP Interpreter via Dataflow Fusion. arXiv:2410.21713 [cs.CR] https://arxiv.org/abs/2410.21713
[43] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM, Pittsburgh PA USA, 386–399. https://doi.org/10.1145/2814270.2814319
[44] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176
[45] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*. Association for Computing Machinery, New York, NY, USA, 1645–1659. https://doi.org/10.1145/3576915.3623166
[46] 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'20)*. 1629–1642. https://doi.org/10.1109/SP40000.2020.00067
[47] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (Sept. 2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681
[48] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th USENIX Security Symposium (Security'21)*. 2795–2809.
[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (ATC'12)*. 309–318.
[50] 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 (ISSTA'21)*. ACM, Virtual Denmark, 244–256. https://doi.org/10.1145/3460319.3464814
[51] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. Association for Computing Machinery, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038
[52] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *2022 USENIX Annual Technical Conference (ATC'22)*. 351–366.
[53] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. 724–735. https://doi.org/10.1109/ICSE.2019.00081
[54] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. (2023).
[55] 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. https://doi.org/10.1145/1993316.1993532
[56] Qi Deng Yiheng An, Haozhe Zhang. [n. d.]. Vulnerabilities in LangChain Gen AI — unit42.paloaltonetworks.com. https://unit42.paloaltonetworks.com/langchain-vulnerabilities/. [Accessed 19-12-2024].
[57] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (Security'20)*. 2307–2324.
[58] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISAN: Sanitizing API Usages through Semantic Cross-checking. (2016).
[59] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, Barcelona Spain, 347–361. https://doi.org/10.1145/3062341.3062379

A Reflection Wrapper Implementation

All reflection information for reflection wrappers directly corresponds to specific reflection calls. Table 7 shows the reflection calls of each language to provision the corresponding reflection information.

Wrapper		Language	
		JavaScript	Python
`enum()`		Object.getOwnPropertyNames(†)	dir(_builtins_)
`refl(o)`	Type	o.constructor.name	type(o)
	Attrs	Object.getOwnPropertyNames(o)	dir(o)
	Arity	o.length	inspect.signature(o)
	Call?	typeof o == function"	callable(o)
Wrapper		PHP	Ruby
`enum()`		get_defined_ {functions,constants,vars}()	global_variables, Object.constants
`refl(o)`	Type	get_class(o)	o.class
	Attrs	get_object_vars(o)	o.{method,class.constants}
	Arity	ReflectionFunction(o)	o.parameters
	Call?	is_callable(o)	o.respond_to? :call

Table 7: Utilized reflection APIs for reflection wrappers in each language. †: `globalThis`.

B Adapting Context-dependent Python Grammar

Note that the syntax of Python is not context-free due to the indentation rule. This can be resolved at the lexical stage by introducing two special tokens: `INDENT` and `DEDENT`, which act similarly to curly braces that enclose the indented block. We modify the Python grammar to produce `INDENT` and `DEDENT` verbatim in the generated code and then augment our fuzzing driver to replace these two tokens with the correct indentation before passing to the interpreter for execution.

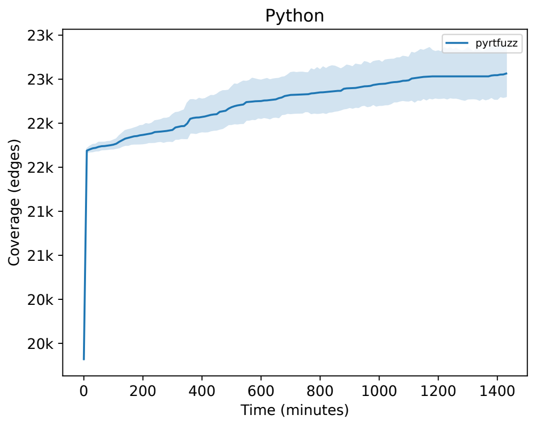
C Extending Reflecta to more targets

Here we demonstrate how easy it is to integrate JavaScriptCore and SpiderMonkey. As mentioned before, Reflecta requires only two pieces of minor, non-increasing manual effort: core syntax and reflection wrapper. Fortunately, V8, JavaScript, and SpiderMonkey implement the same ECMA-262 language specification [4], meaning both the core syntax and reflection wrapper could be used without modification. Only minor changes were done to the harness (~ 30 loc), so that the generated program could write reflection information to a memory-backed file. No other modification was needed for the fuzzer, and integration was done in an afternoon. In the end, Reflecta achieved 13.54% and 14.21% edge coverage for JavaScriptCore and SpiderMonkey respectively (starting from scratch and repeated 8 times for 24 hours). Reflecta adapts to different implementations with little effort, with zero changes to the core fuzzing loop. Note when extending to different languages, the manual effort is also low, from Table 3 we anticipate around 80 loc of addition of the core syntax, and implementing two wrapper function in the target language. This highlights Reflecta's capability to *scale* to multiple language with *minimal manual effort.*

D Reproducing bugs found by PyRTFuzz

In an attempt to reproduce the bugs found by PyRTFuzz, we first analyzed its bug reports and discovered inconsistencies. We noted that the majority of the bugs (44 out of 61 bugs reported) reports by PyRTFuzz are unhandled exceptions raised by the interpreter. Such reports are not considered bugs because the interpreter exits gracefully when an exception is thrown and is considered the expected behavior [8, 13]. Of the remaining 17 reports, there are 7 out-of-memory errors, 5 hangups and 5 stack overrun, which may be considered as DoS vulnerabilities but have limited exploitability. We observed that all bug reports of PyRTFuzz have been closed by the CPython developers. In contrast, reflecta does count them as bugs and only report testcases that are terminated by signal SIGSEGV to the developers. The global-buffer-overflow, heap-buffer-overflows, and use-after-frees we discovered are potentially exploitable and can be used to launch remote code execution attacks in a sandboxed environment.

E Reproducing PyRTFuzz's coverage



Note in PyRTFuzz's paper, the authors did not detail how they measured coverage. We follow the stand practice of replaying the corpus after campaign and measure edges covered in CPython using a SanitizerCoverage instrumented binary. The experiment is repeated 8 times and lasted 24 hours (in line with other experiments in the evaluation). Our reproduced coverage growth shows that most of CPython's code are covered by the generated initial seeds, and mutation did not significantly improve coverage (from 22k edges to 23k edges), which deviates from Figure 8 in PyRTFuzz's paper.

F Extra Case Studies

Here we showcase more case studies to demonstrate that reflecta can find various types of bugs in Micorpython, PHP, and MRuby.

Figure 10 shows the minimized proof-of-concept program for the global buffer overflow bug at `task_push_queue` in MicroPython. The program simply attempts to push an item to the `waiting` queue of an `asyncio.Event` instance, which eventually causes a global buffer overflow when updating the queue.

The uniqueness of this bug is that the `waiting` method of an `asyncio.Event` instance is neither present in CPython (reference impl.) nor part of the public API, which explains why this simple bug has evaded the unit test suite covering 98.5% [6] of the entire

```
import asyncio, io
v1 = asyncio.Event();
v2 = v1.waiting;
v2.push(io);
```

Figure 10: Minimized proof-of-concept program for the global buffer overflow bug at `task_push_queue` in MicroPython.

MicroPython source code. Furthermore, this bug also reconfirms the completeness concern of manual specifications. Particularly, since the `waiting` method of an `asyncio.Event` is both MicroPython-specific and outside of the well-documented public APIs, any manual grammar specifications *not* tailored to MicroPython or *only* considering documented features are destined to miss this bug.

```
v1 = Range.methods();
v1.shift();
v1.rotate!();
```

Figure 11: Minimized proof-of-concept program for the use-after-free bug at `ary_rotate_bang` in MRuby

The root cause of this bug stems from an internal space optimization in MRuby. When shifting an array larger than 20 elements (as returned by `Range.methods()`), MRuby opts to return a slice of the original array and marks it as `SHARED`, rather than physically moving all elements one-by-one. The `rotate!` method, however, attempts to rotate the array in place. To do this, it must create a private copy and may free the old buffer if there is only a single reference to the slice. Due to a developer oversight, the old buffer pointer—now freed—was still used after the new private copy was created, causing a use-after-free. Although the proof-of-concept is straightforward, both NAUTILUS and POLYGLOT failed to trigger this bug because their grammars included `rotate` but not `rotate!`. This omission likely resulted from imperfect documentation scraping.

```
// global enumeration
$v0 = get_defined_functions();
$v1 = $v0[283]
$v2 = $v0[3]
// is_callable($v1): true
// class_of($v2): "string"
$v0($v2);
```

Figure 12: Minimized proof-of-concept program for the null-dereference bug at `zif_func_num_args` in PHP

The root cause of this PHP bug lies in the assumption that `func_get_args` is always called inside a user-defined variadic function to determine the number of arguments passed. Under normal circumstances, this function expects an outer call frame stored in the global variable `EX(prev_execute_data)`. However, when it is invoked by the runtime via `register_shutdown_function`—which does not create an outer call frame—a null-dereference occurs due to the missing context.

REFLECTA uncovered this bug by first enumerating all defined functions using `get_defined_functions`, then recording the string `"func_get_arg"`, and finally discovering that `register_shutdown_function` is callable and accepts strings as arguments. Interestingly, in PHP, a string can be called as a function if its contents match a defined function name. While NAUTILUS includes both `func_get_arg` and `unregister_shutdown_function` in its grammar, it cannot produce this specific bug because it only allows direct function calls like `unregister_shutdown_function(...)` and not string literals such as `"unregister_shutdown_function"`. Similarly, POLYGLOT failed to expose the bug because neither function was available through its grammar or semantics. By contrast, reflecta not only discovered these functions via global enumeration but also realized that one is callable and the other is a string, enabling it to trigger the bug.

G Grammar Snippets

$\langle program \rangle$ ::= $\langle statement \rangle^*$

$\langle var \rangle$ ::= v[0-9]+

$\langle statement \rangle$ ::= $\langle loadInteger \rangle$ | $\langle loadBigInt \rangle$ | $\langle loadFloat \rangle$ | $\langle loadString \rangle$ | $\langle loadBoolean \rangle$ | $\langle loadBuiltin \rangle$ | $\langle getProperty \rangle$ | $\langle callFunction \rangle$ | $\langle callMethod \rangle$ | $\langle unaryOperation \rangle$ | $\langle binaryOperation \rangle$ | $\langle ternaryOperation \rangle$ | $\langle reassign \rangle$ | $\langle update \rangle$ | $\langle dup \rangle$ | $\langle compare \rangle$ | $\langle eval \rangle$ | $\langle print \rangle$

$\langle binaryOperation \rangle$ ::= $\langle var \rangle$ $\langle operator \rangle$ $\langle var \rangle$

$\langle operator \rangle$ ::= + | - | * | / | % | & | && | || | ^ | « | » | **

Figure 13: Conceptual core grammar of Reflecta. Statements are implemented as FuzzIL instructions, and the list of supported IL instructions are listed below. Representation and lifting of IL instructions can be found here: https://github.com/HexHive/Reflecta

Currently, Reflecta implements minimal grammars in Fuzzilli lifters using a subset of Fuzzilli IR opcodes. Figure 14 shows the list of implemented opcodes.

```
    case .loadInteger(let op):
    case .loadBigInt(let op):
    case .loadFloat(let op):
    case .loadString(let op):
    case .loadBoolean(let op):
    case .loadBuiltin(let op):
    case .getProperty(let op):
    case .callFunction:
    case .callMethod(let op):
    case .unaryOperation(let op):
    case .binaryOperation(let op):
    case .ternaryOperation:
    case .reassign:
    case .update(let op):
    case .dup:
    case .compare(let op):
    case .eval(let op):
    case .print:
```

Figure 14: Implemented Fuzzilli IR opcodes.

```
ctx.rule(u'START',u'<?php\n$a = NULL;\n$b = NULL;\n$c = NULL;\
      n$d = NULL;\nsrand(1337);\n{PROGRAM}?>')
ctx.rule(u'PROGRAM',u'{STATEMENT};\n{PROGRAM}')
ctx.rule(u'PROGRAM',u'')
ctx.rule(u'STATEMENT',u'function {FUNCTION}({ARGS})\n{PROGRAM
      }}')
ctx.rule(u'STATEMENT',u'{VAR} = {FUNCTION}({ARGS})')
ctx.rule(u'STATEMENT',u'{VAR} = {VAR}->{FUNCTION}({ARGS})')
ctx.rule(u'STATEMENT',u'{VAR} = {CLASS}->{FUNCTION}({ARGS})')
ctx.rule(u'STATEMENT',u'{VAR} = {VAL}')
ctx.rule(u'STATEMENT',u'return {VAR}')
ctx.rule(u'STATEMENT',u'raise {VAR}')
ctx.rule(u'STATEMENT',u'yield {VAR}')
ctx.rule(u'STATEMENT',u'continue {VAR}')
ctx.rule(u'STATEMENT',u'break {VAR}')
ctx.rule(u'STATEMENT',u'next {VAR}')
```

Figure 15: Snippets of Nautilus's PHP grammar, the full grammar can be found at https://github.com/nautilus-fuzz/nautilus/blob/mit-main/grammars/php_custom.py

```
parser grammar PhpParser;

options { tokenVocab=PhpLexer; contextSuperClass =
      PolyGlotRuleContext; }

@parser::header {
#include "polyglot_rule_context.h"
}

program
    : Shebang? (inlineHtml | phpBlock)* EOF
    ;

inlineHtml
    : htmlElement+
    | scriptText
    ;

htmlElement
    : HtmlDtd
    | HtmlClose
    | HtmlStyleOpen
    | HtmlOpen
    | HtmlName
    | HtmlSlashClose
    | HtmlSlash
    | HtmlText
    | HtmlEquals
    | HtmlStartQuoteString
    | HtmlEndQuoteString
    | HtmlStartDoubleQuoteString
    | HtmlEndDoubleQuoteString
    | HtmlHex
    | HtmlDecimal
    | HtmlQuoteString
    | HtmlDoubleQuoteString
```

Figure 16: Snippets of PolyGlot's PHP grammar, the full grammar can be found at https://github.com/OMH4ck/PolyGlot-Grammar/blob/main/php/PhpParser.g4