One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation

Yongheng Chen*, Rui Zhong†, Hong Hu†, Hangfan Zhang†, Yupeng Yang‡, Dinghao Wu† and Wenke Lee*
*Georgia Institute of Technology, †Penn State University, ‡UESTC

***Abstract*—Language processors, such as compilers and interpreters, are indispensable in building modern software. Errors in language processors can lead to severe consequences, like incorrect functionalities or even malicious attacks. However, it is not trivial to automatically test language processors to find bugs. Existing testing methods (or fuzzers) either fail to generate high-quality (i.e., semantically correct) test cases, or only support limited programming languages.**

In this paper, we propose PolyGlot, a generic fuzzing framework that generates high-quality test cases for exploring processors of different programming languages. To achieve the generic applicability, PolyGlot neutralizes the difference in syntax and semantics of programming languages with a uniform immediate representation (IR). To improve the language validity, PolyGlot performs constrained mutation and semantic validation to preserve syntactic correctness and fix semantic errors. We have applied PolyGlot on 21 popular language processors of 9 programming languages, and identified 173 new bugs, 113 of which are fixed with 18 CVEs assigned. Our experiments show that PolyGlot can support a wide range of programming languages, and outperforms existing fuzzers with up to 30× improvement in code coverage.

I. Introduction

Language processors [70], such as compilers and interpreters, are indispensable for building modern software. They translate programs written in high-level languages to low-level machine code that can be understood and executed by hardware. The correctness of language processors guarantees the consistency between the semantics of the source program and the compiled target code. Buggy language processors can translate even correct programs to malfunctional codes, which might lead to security holes. For example, miscompilation of memory-safe programs produces memory-unsafe binaries [14, 15]; vulnerabilities in interpreters enable attackers to achieve denial-of-service (DoS), sandbox escape, or remote code execution (RCE) [6, 53, 57]. Even worse, these defects affect all translated programs, including other translated language processors [59].

However, it is nontrivial for traditional software testing techniques to automatically detect bugs in language processors, as the processors pose strict requirements on their inputs regarding the syntactic and semantic validity. Any error in the program can terminate the execution of the language processor and hinder the tester from reaching deep translation logic.

Recent works on software testing, such as grey-box fuzzing, try to meet these requirements to effectively test language processors [17, 26, 38, 48, 76, 77]. Originally, structure-unaware mutation [38, 76, 77] can hardly generate syntax-correct test cases; advanced fuzzers [42, 44] adopt higher-level mutation in the abstract syntax tree (AST) or the immediate representation (IR) to preserve the input structures. Alternatively, generation-based fuzzers leverage a precise model to describe the input structure [1, 2, 51], and thus can produce syntax-correct test cases from scratch. To further improve the semantic correctness, recent fuzzers adopt highly specialized analyses for specific languages [53, 74, 78].

However, a fuzzer will lose its generic applicability when it is highly customized for one specific language. Users cannot easily utilize the specialized fuzzer to test a different programming language, but have to develop another one from scratch. Considering the complexity of language-specific fuzzers (*e.g.*, `CSmith` [74] consists of 80k lines of code) and the large number of programming languages (over 700 currently [3]), it is impractical to implement a specific fuzzer for each language. This puts current fuzzers in a dilemma: pursuing high semantic validity sacrifices their generic applicability, while retaining generic applicability cannot guarantee the quality of test cases.

In this paper, we propose PolyGlot, a fuzzing framework that can generate *semantically valid* test cases to extensively test processors of *different* programming languages. To achieve generic applicability, we design a uniform IR to neutralize the difference in the syntax and semantics of programming languages. Given the BNF (Backus–Naur form) grammar [66] of a language, PolyGlot can generate a frontend that translates source programs into this IR. At the same time, users can provide semantic annotations to describe the specific semantics about the scopes and types of definitions of the language. The definitions include the defined variables, functions, and composite types. In this paper, we use variables and definitions interchangeably. These annotations will produce semantic properties in IR during translation. For example, the BNF grammar of functions in C is `<func := ret-type func-name arg-list func-body>`. We can give annotations such as "`func` defines a new function" and "`func-body` creates a new scope". In this way, the language differences of programming languages are unified in the IR.

To achieve high language validity, we develop two techniques, the *constrained mutation* and the *semantic validation*, for test case generation. The constrained mutation retains the grammar structure of the mutated test cases, which helps preserve their syntactic correctness. Further, it tries to maintain the semantic correctness of the unmutated part of the test case. For example, it avoids mutating the statement "`int x = 1;`" in a C program in case that the rest of the program uses `x`, which otherwise introduces the error of using undefined variables.

Since the syntactic correctness of a test case is preserved, and the semantic correctness of the unmutated part is still valid, the only thing left is to fix the potential semantic errors in the mutated part. The mutated part could introduce semantic errors because it brings in new code segments, which might use variables that are invalid in the mutated test case. To fix the errors, we replace these invalid variables according to the rules of scopes and types. For example, our mutation may insert a statement "`unknownVar + 1;`" into the mutated program P which does not define `unknownVar`. Assuming P defines two variables, `num` of type integer and `arr` of type array, we should replace `unknownVar` with `num` because addition by 1 requires the variable to have an integer type. Our semantic validation utilizes the semantic properties of IR to collect type and scope information of variables in the test case and integrates them in the symbol tables. These symbol tables describe the types, scopes and the names of every variable. The semantic validation then utilizes them to replace invalid variables with valid ones in the mutated code, which greatly improves the semantic correctness (up to 6.4× improvement in our evaluation §VIII-C).

We implement PolyGlot with 7,016 lines of C++ and Python codes, which focus on IR generation, constrained mutation and semantic validation. PolyGlot currently supports 9 programming languages and we can easily adopt it to others.

We evaluate PolyGlot on 21 popular compilers and interpreters of 9 different programming languages and successfully find 173 new bugs. At the time of paper writing, 113 of the bugs have been fixed with 18 CVEs assigned. Our experiments show that PolyGlot is more effective in generating high-quality test cases (up to 100× improvement in language validity), exploring program states (up to 30× more new paths) and detecting vulnerabilities (8× more unique bugs) than state-of-the-art general-purpose fuzzers, including the mutation-based fuzzer `AFL`, the hybrid fuzzer `QSYM` and the grammar fuzzer `Nautilus`. We also compare PolyGlot with language-specific testing tools `CSmith` for C and `DIE` for `JavaScript`, whose results show that PolyGlot can explore more program states.

In summary, this paper makes the following contributions:

- We propose a generic framework that can produce high-quality inputs to test different language processors.
- We implement the prototype, PolyGlot, of our system to effectively test language processors.
- We evaluate PolyGlot on 21 language processors of 9 programming languages and identify 173 new bugs.

We will release the source code of PolyGlot.

II. Problem

In this section, we first briefly describe how language processors handle input programs, and how syntax errors and semantic errors terminate this process. Next, we illustrate the challenges and limitations of existing fuzzers in testing language processors. Then, we summarize the common semantic errors in test cases generated by fuzzing tools. Finally, we present our insights to solve this problem.

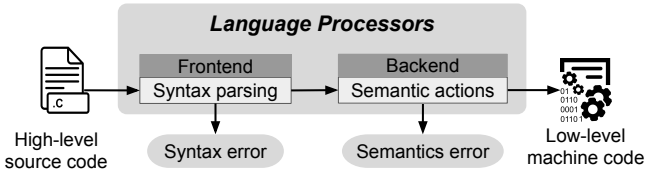


Fig. 1: Workflow of language processors. Given a high-level source-code program, a language processor checks it for syntactic and semantic errors. If none, the processor translates the program into low-level representations.

```
1  struct S { int d; } s;
2  int a, c;
3  int main() {
4    short *e, b[3] = {1, 2, 0};
5    e = b;
6  + e = b    // missing ';'
7  + e = s;   // mismatch type
8  + e = res; // undef var
9    do{c += *(e++);} while(*e);
10   int res = c;
11   return res;
12 }
```

(a) An example C program

```
1   function opt(x){return x[1];}
2   let arr = [1, 2];
3
4   if(arr[0]) let arr2=[1, 2];
5   // ig is a wrong keyword
6 + ig(arr[0]) let arr2=[1, 2];
7   arr[1] += "1234";
8
9   for(let idx=0; idx<100; idx++)
10    opt(arr);
11 + for(let idx=0; idx<100; idx++)
12 +   opt(arr2); // undef var
```

(b) An example `JavaScript` program

Fig. 2: Running examples. Fig. 2a shows a program written in C, a statically typed programming language. If we replace line 5 with one of line 6–8, we get different errors as stated in the comments. Similarly, Fig. 2b shows a program written in `JavaScript`, a dynamically typed language, which allows more type conversion.

A. Language Processors

Language processors convert programs written in high-level languages into low-level machine codes. For example, compilers translate the whole program into machine codes, while interpreters translate one statement at a time.

Language processors check the input program for both syntactic and semantic errors. Any error can terminate the execution of processors. We show the workflow of language processors in Fig. 1. The frontend checks for syntax errors at the early stage of processing. Afterward, the backend checks for semantic errors, which cannot be detected by the parser. Only semantically correct programs can be successfully processed.

We show a C program in Fig. 2a and a `JavaScript` program in Fig. 2b. If we add the statements that start with "+", we introduce errors in the program. For example, line 6 in Fig. 2a and line 6 in Fig. 2b introduce syntax errors, and the parser detects these errors and bails out. Line 7–8 in Fig. 2a and line 11–12 in Fig. 2b contain semantic errors which will be caught by the backend optimizer or translator.

B. Limitations of Current Fuzzers

Fuzzing is a well-received technique of discovering bugs and vulnerabilities in software [7, 17, 77]. However, current fuzzers have limitations in testing language processors. General-purpose mutation-based fuzzers [30, 38, 76, 77] are unaware of input structures and randomly flip the bits or bytes of the inputs, so they can hardly generate syntax-correct inputs. Recent works adopt higher level mutation in AST or IR to guarantee the syntactic correctness [44, 63]. Alternatively, generation-based fuzzers [1, 51] utilize a model or grammar to generate structural inputs effectively. These fuzzers have shown their advantages in

passing the syntactic checks over random bitflip mutation-based fuzzers. Yet they ignore the semantic correctness of generated test cases and fail to find deep bugs in the optimization or execution code of language processors. We did a quick test to understand how semantic correctness helps fuzzers reach deeper logic: compiling the code in Fig. 2a with "-O3" covers 56,725 branches in gcc-10, while those invalid variants starting with "+" only trigger less than 27,000 branches as they are rejected during or right after the parsing.

Researchers try to specialize their fuzzers for higher semantic correctness [42, 53, 74, 78]. CSmith [74] performs heavy analyses to generate valid C programs without undefined behaviors. `JavaScript` fuzzers [42, 53] consider the types of expressions to avoid semantic errors in generated test cases. `Squirrel` [78] tackles the data dependency of SQL to generate valid queries to test DBMSs. Unfortunately, these approaches are highly specialized for one programming language. Users need to put huge development efforts to adopt them on new programming languages, which is time-consuming and impractical considering the large number of real-world languages [3].

Recent language-based fuzzers [22, 44] try to generate correct test cases for different languages. LangFuzz [44] replaces every variable in the mutated code randomly while `Nautilus` [22] uses a small set of predefined variable names and relies on feedback guidance to improve semantic correctness. However, these strategies are only effective in testing languages which allow more implicit type conversion, such as `JavaScript` and PHP.

C. Common Semantic Errors

We manually investigate 1,500 invalid test cases generated by existing language fuzzers [22, 51, 53] and summarize four common types of semantic errors. Two of them are related to the scope of variables and functions, and the rest two are related to the types of variables and expressions. These errors violate the common rules of types and scopes on definitions and are language-independent.

Undefined Variables or Functions. Variables or functions should be defined before they can be used. Otherwise, the behaviors of the program can be undefined or illegitimate. For example, line 8 of Fig. 2a uses an undefined variable `res` and C compilers refuse to compile the code.

Out-of-scope Variables or Functions. In a program, variables or functions have their scopes, which determine their visibility. We cannot use an out-of-scope invisible variable or function. For example, `arr` is visible at line 10 of Fig. 2b, while `arr2` is not since its scope is within the `if` statement at line 4.

Undefined Types. Many programming languages allow users to define custom types, such as `class` in `JavaScript` and `struct` in C. Like variables, such types should be defined before their instances can be used.

Unmatched Types. Usually, assigning a value to a variable of incompatible type or comparing incompatible types introduces semantic errors. In some cases, programming languages allow type conversions, which convert mismatched types to compatible ones explicitly or implicitly. For example, `e` of type `pointer of short` and `s` of type `S` are not compatible in C, so line 7 of Fig. 2a introduces an error. Line 7 of Fig. 2b is correct because in `JavaScript` numbers can convert to strings.

D. Our Approach

The goal of this paper is to build a generic fuzzing framework that generates semantically correct inputs to test different language processors. We achieve the goal in two steps. First, we neutralize the difference in syntax and semantics of programming languages by embedding them into a uniform IR, so we can perform uniform mutation or analysis regardless of the underlying languages. Second, we constrain our mutation to generate new test cases, which might contain semantic errors, and then we perform semantic validation to fix these errors.

Neutralizing Difference in Programming Languages. Different programming languages have unique syntax and semantics. To neutralize their differences, we design a new immediate representation to map the language-specific features, both syntactic and semantic, into a uniform format. Given the BNF grammar of a language, we can generate a frontend to translate a source program into an IR program. The IR program consists of a list of IR statements or IRs as we call them in this paper. This IR program keeps the syntactic structures of source programs so we can easily translate it back into the original source. Regarding semantics, we design a simple annotation format for users to describe the scopes and types of a language. These descriptions will be encoded into the semantic properties of the IR and guide our system to fix semantic errors. After the language-specific grammar and semantics are captured by the IR, we can perform mutation or analyses regardless of the underlying language.

Improving Language Validity. We improve the language validity with *constrained mutation* and *semantic validation*. Our constrained mutation tries to preserve two aspects of the program: the syntactic correctness of the whole program, and the semantic correctness of the unmutated part. First, we mutate the IRs based on their IR types that reflect the underlying grammar structures. This preserves the syntactic correctness of the test case. For example, we replace an `IF` statement (in the form of IR) with another `IF` statement instead of a function call expression. Second, we only mutate IRs with *local effects* to preserve the semantic validity of the unmutated code. Such IRs contain no definitions or create new local scopes. For example, in Fig. 2b, line 7 has local effects because it only uses the variable `arr` and contains no new definition. Without line 7, the rest of the program is still valid. Therefore, assuming the initial test case is correct, a mutated variant produced by constrained mutation only has potential semantic errors in the mutated part, which might use invalid variables. To fix these errors in a systematic way, our semantic validation first utilizes IR's semantic properties to collect type and scope information of the mutated test case. We integrate the collected information into the symbol tables, which contain the types, scopes and names of every definition. These symbol tables guide PolyGlot to replace the invalid use of variables properly. Afterward,

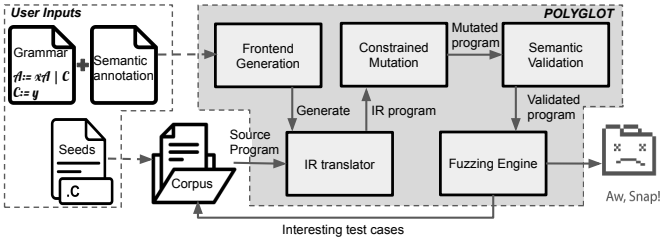


Fig. 3: Overview of PolyGlot. PolyGlot aims to discover bugs that crash language processors. PolyGlot accepts the BNF grammar, semantic annotations, and seeds from users as input. First, the frontend generator generates an IR translator that converts a source program to an IR program. Second, the constrained mutator mutates the IR program to get new ones, which might contain semantic errors. Next, the semantic validator fixes the semantic errors. Finally, the fuzzer runs validated programs to detect bugs.

the validated test cases should be correct and are helpful for thoroughly fuzzing language processors.

III. Overview of PolyGlot

Fig. 3 shows an overview of PolyGlot. Given the BNF grammar, semantic annotations and initial test cases of the targeted programming language, PolyGlot aims to find inputs that trigger crashes in the language processor. First, the frontend generator generates an IR translator using the BNF grammar and the semantic annotations (§IV). Then, for each round of fuzzing, we pick one input from the corpus. The IR translator lifts this input into an IR program. Next, the constrained mutator mutates the IR program to produce new syntax-correct ones, which might contain semantic errors (§V). Afterward, the semantic validator tries to fix the semantic errors in the new IR programs (§VI). Finally, the IR program is converted back to the form of source code and fed into the fuzzing engine. If the test case triggers a crash, we successfully find a bug. Otherwise, we save the test case to the corpus for further mutation if it triggers a new execution path.

IV. Frontend Generation

To achieve generic applicability, our frontend generation generates a translator that transforms a source program into an IR program. This lowers the level of mutation and analysis from language-specific source code to a uniform IR.

In Fig. 4, we show the IR (Fig. 4a) of a simple C program to demonstrate how the BNF grammar (Fig. 4b) and semantic annotations (Fig. 4c) help construct the IR statements. Each symbol in the BNF grammar generates IRs of a unique type (*e.g.*, symbol `<func-def>` generates `ir9` of type `FuncDef`). The original source code is stored in the `op` or `val` of the IRs (*e.g.*, the `val` of `ir2` stores the name `main`). We predefine semantic properties about types and scopes of variables for users to use. The generated IRs will carry these properties as described in the annotations (*e.g.*, `ir9` has property `FunctionDefinition`). Users can easily use the BNF grammar and semantic annotations to describe the specific syntax and semantics of a programming language.

A. Intermediate Representation

Our IR is in a uniform format and captures the syntax and semantics of the source program. It includes an order, a type, an operator, no more than two operands, a value, and a list of semantic properties. The IR order and the type correspond to the statement order in source code and the symbol in the BNF grammar respectively. The IR operator and the IR value store the original source code. All the IRs are connected by the IR operands, which are also IRs. These parts carry the syntactic structures, while the semantic properties describe the semantics of the source program, as discussed below.

Syntactic Structures. Syntactic structures keep all the grammar information of the source program. As we see earlier, some IRs store a small piece of the source code (*e.g.*, a function name is stored in an IR of type `FuncName`). Also, the IRs are connected in a directed way that forms a tree view of the source program. If we perform inorder traversal on the IR program, we can reconstruct the original source program.

Semantic Properties. Semantic properties capture the semantics about the scopes and types of definitions. They tell us which IRs belong to variable definitions. Additionally, for scopes, they tell which IRs create new scopes so that we can decide the visibility of the variables within the scopes. For types, they describe the predefined and user-defined types in a language and their type conversion rules. They also describe the expected operand types and the output type of operators, which perform mathematical, relational, or logical operations and produce a result.

For example, `ir9` in Fig. 4a has the semantic property `FunctionDefinition`, indicating that it relates to a function definition. Line 14-15 in Fig. 4c describe the inference rule for the operator "+", which accepts two operands of type `long` and outputs a result of type `long`. Assuming number literals are of type `long`, we know `11 + 12` produces a result of `long`.

B. Generating IR Translator

To generate a translator, users should provide the BNF grammar, which describes the unique syntax, and semantic annotations, which capture the specific semantics of a language. The language information of these two files is embedded into the syntactic structures and semantic properties of IR respectively. First, the frontend generator treats every different symbol in the grammar as a different object. Then it analyzes the semantic annotations to decide which symbols should have what semantic properties. Finally, it generates for each object unique parsing and translation methods, which parse the source code and generate IRs with required semantic properties. These generated methods composite an IR translator.

V. Constrained Mutation

As the first step towards language validity, we apply two rules to constrain our mutation on an initially correct test case to preserve its syntactic correctness (§V-A) and the semantic correctness of its unmutated part (§V-B). The former is the base for semantic correctness, and the latter makes it possible

```
//IR<type,       left, right, op,          val
//       [, semantic_property]>
ir0<Type,       NIL,  NIL,   NIL,        "int">
ir1<RetType,    ir0,  NIL,   NIL,         NIL >
ir2<FuncName,   NIL,  NIL,   NIL,       "main",
        [FunctionName]>
ir3<Literal,    NIL,  NIL,   NIL,          12 >
ir4<Literal,    NIL,  NIL,   NIL,          23 >
ir5<BinaryExpr, ir3,  ir4,   "+",         NIL >
ir6<RetStmt,    ir5,  NIL,   "return ;", NIL >
ir7<FuncBody,   ir6,  NIL,   "{ }",       NIL,
        [FunctionBody, NewScope]>
ir8<NIL,        ir1,  ir2,   "()",        NIL >
ir9<FuncDef,    ir8,  ir7,   NIL,         NIL,
        [FunctionDefinition]>
ir10<Program,   ir9,  NIL,   NIL,         NIL >
```

(a) IR program for "`int main() { return 12+23;}`". The format of IR is shown in the comments at the top.

```
<program> ::=
  (<global-def> | <func-def>)*
...

<func-def> ::=
  (<ret-type> <func-name>
  "(" <func-arg>? ")" <func-body>)

<ret-type> ::= <type>

<type> ::=
  ("int" | "short" | ...)

<binary-expr> ::=
  <literal> "+" <literal>
...
```

(b) Part of the BNF grammar for `C` programs.

```
{ "Comment1": "Scopes and composite types",
  "func-def":["FunctionDefinition"],
  "func-name":["FunctionName"],
  "func-body":["FunctionBody", "NewScope"],

  "Comment2": "Types and conversion rules",
  "BasicType": ["int", "short", "..."],
  "ConversionRule": [
    {"short": ["int", "..."]}
  ],

  "Comment3": "Type inference rules",
  "TypeInference":{
    "+": { "left": "long", "right": "long",
      "output":"long"} }
}
```

(c) Part of semantic annotations for the grammar in Fig. 4b. It is in `JSON` format.

Fig. 4: An example IR program with its corresponding BNF grammar and semantic annotations. The IRs in Fig. 4a are in a uniform format. The number suffix of the IR is the IR order. The IR types have corresponding symbols in the BNF grammar in Fig. 4b. `left` and `right` are the two operands. As `func-def` has four components and the IR can have no more than two operands, `ir8` is an intermediate IR for `func-def` and does not have a type. The semantic annotations in Fig. 4c describe what semantic properties the symbols in the BNF should have, which will be reflected in the `semantic_property` in IRs. We predefine properties about types and scopes for users to use.

to gain language validity by fixing the semantic errors in the mutated part.

A. Rule 1: Type-Based Mutation

This rule performs three different mutation strategies based on the IR types. *Insertion* inserts a new IR (*e.g.*, IRs from another program) to the IR program. This includes inserting an IR that represents an element to a list and inserting an IR to where it is optional but currently absent. *Deletion* performs the opposite operation of insertion. For example, in C, a statement block is a list of statements, and we can insert a statement to the block. Also, we can delete an optional `ELSE` statement after an `IF` statement. *Replacement* replaces an IR with a new one of the same type. For example, we can replace an addition expression with a division expression as they are both of type `EXPRESSION`.

Since the IR type reflects the grammar structures of underlying source codes, this rule helps preserve the syntactic correctness of the mutated test cases.

B. Rule 2: Local Mutation

This rule requires us to only mutate IRs with local effects. Changes in these IRs will not invalidate the semantic correctness of the rest of the program. POLYGLOT handles two types of IRs with local effects as follows.

IRs that Contain No New Definitions. These IRs do not define any variables, functions, or types, so the rest of the program will not use anything defined by them. Even if these IRs get deleted, the rest of the program will not be affected. For example, line 7 in Fig. 2b only uses the variable `arr` and does not define anything. If we delete this line, the program can still be executed.

IRs that Create Scopes. These IRs can contain new definitions, but these definitions are only visible within the scope created by the IRs. For example, the `for` statement at line 9 of Fig. 2b creates a new scope. `idx` is defined and only valid within this new local scope. Therefore, mutating the `for` statement as a whole will not affect the rest of the program.

With these two rules, our constrained mutation produces syntax-correct test cases. These test cases might contain semantic errors. According to local mutation, the semantic errors are introduced by the mutated part, which might use invalid variables from other test cases. Next, we will fix all these errors to get a semantically correct test case.

VI. SEMANTIC VALIDATION

As the second step towards language validity, we perform semantic validation to fix the semantic errors in the mutated part of the test case. We do so by replacing the invalid variables with the valid ones. To figure out the proper variables for replacement, we first need to know the scopes of the variables, which tell us the available variables to use. This avoids using undefined or out-of-scope variables. Further, we need to know the types of these variables so that we can use them appropriately, which avoids using variables of undefined or unmatched types.

Therefore, our semantic validation relies on two components: *type system* that collects type information of variables (§VI-A) and *scope system* that collects scope information (§VI-B). We then integrate all the information into the symbol tables, which contain the types, scopes, and names of every definition in the test case. With the symbol tables, the semantic validation generates correct expressions to replace the invalid ones and produces a semantically correct test case (§VI-C).

A. Type System

In programming languages, types include predefined ones such as `int` in C, and user-defined ones such as `class` in `JavaScript` [8]. We call the former *basic types* and the latter *composite types*. Basic types are limited so they can be completely described with semantic annotations, but composite types cannot as they are specific to test cases. To collect precise type information, we need to handle both basic types and composite types. Therefore, POLYGLOT utilizes the type system to construct composite types on demand and infer types of variables or expressions.

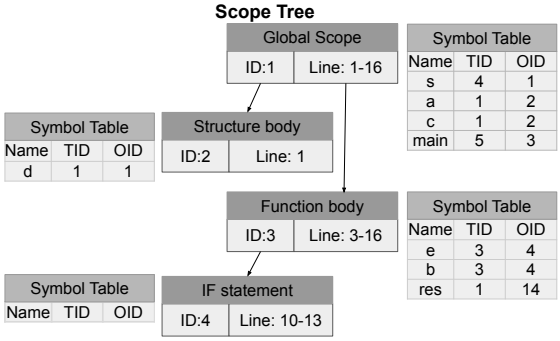
```
1  struct S { int d; } s;
2  int a, c;
3  int main() {
4    short *e, b[3] = {1, 2, 0};
5    e = b;
6    // Originally: do{ c += *(e++); } while(*e);
7    /* Replaced by: if(x >= y){
8                      struct X z;
9                      x += y; } */
10   if(FIXME >= FIXME){
11   //struct X z;
12     FIXME += FIXME;
13   }
14   int res = c;
15   return res;
16 }
```

(a) A mutated program that needs to be validated

Type map

TID	Type
1	int
2	short
3	short *
4	Type:struct Name: S Member: int d
5	Type: function Name: main Arg: None Return type: int

Scope Tree



(b) The type map, scope tree and its the symbol tables of the program in Fig. 5a

Fig. 5: A mutated variant of Fig. 2a and its semantic information collected by the semantic validator. The mutated program is generated by replacing the `DO-WHILE` statement (line 6) with `IF` statement (line 10-13) in Fig. 5a. Every invalid variable in the mutated part is replaced with a `FIXME`. Line 11 is removed because it uses a composite type (`struct X`) without a detailed definition. As shown in Fig. 5b, the type map contains the types used in the program. The program has four different scopes created by different symbols. Each scope has a symbol table with the information of definitions within the scope. `TID` in the symbol table refers to the `TID` in the type map. `OID` corresponds to the statement order in the IR program and we currently use line number for easy demonstration.

Type Map. As the collected type information will be used frequently, we maintain them in a type map for easy and fast access. The key of the map is a unique id for the type, and the value is the structure of the type. This map stores all the basic types of a language and the composite types used in the current test case. For example, Fig. 5a and Fig. 5b show a mutated program and its type map. We can see that type id 5 refers to a function type whose name is `main` and return type is `int`. As composite types are specific to a test case, we remove them from the map each time we finish processing a test case to avoid using types defined in other programs.

Composite Type Construction. Currently, the type system supports the construction of three composite types: structures, functions, and pointers. These types consist of several components. For example, a function consists of a function name, function arguments, and return value.

To construct a composite type, the type system walks through the IR program to find IRs related to composite type definitions by checking their semantic properties. When it finds one, it searches for the required components for this definition. Then type system creates a new type with the collected components and stores it in the type map.

Type Inference. The type system infers the types of variables so that we know how to use them correctly. We handle both the variable definition and variable use. For a variable definition, we check whether it has an explicit type. If so, the type system searches the name of the type in the type map. Then it returns the corresponding type id when the names match. Otherwise, we infer the type of a variable from its assigned expression, which will be discussed in the next paragraph. For example, in `C`, `"int y;"` explicitly states that the variable `y` is of type `int`. In `JavaScript`, `"let z = 1.0;"` does not state a type for `z`, but we can infer from expression `1.0` that `z` is of floating-point number type. For variable use, we just look for the variable name in the symbol tables (§VI-B), which contains the type information of variables, and return its type.

To infer the type of an expression, we first check whether it consists of a simple variable or literal. If it is a variable, it must be a variable use, which has been handled above. If it is a literal, we return its type as described in the semantic annotations. For an expression with operands and operator, we first recursively infer the types of the operands as they are also expressions. As discussed in §IV-A, the semantic properties describe the expected operand types and the output type of the operator. If the inferred operand types can match or convert to the expected types, we return the corresponding result type.

Our type inference has limitations in dynamically typed programming languages, where the types might be undecidable statically. For example, the type of `x` in line 1 of Fig. 2b is not determined because `JavaScript` can call the function with arguments of any type. If we simply skip the variables whose types cannot be inferred, we might miss useful variables. Therefore, we define a special type called `AnyType` for these variables. Variables of `AnyType` can be used as variables of any specific type. Using `AnyType` might introduce some type mismatching, but it can improve the effectiveness of POLYGLOT in dynamically typed programming languages.

B. Scope System

A program can have different scopes that decide the visibility of variables within them. The scope system partitions the program into different scopes so that variables automatically gain their visibility according to the scope they are inside. Afterward, we integrate the type information collected by the type system and the scope information into symbol tables. The symbol tables contain all the necessary information of variables for fixing the semantic errors.

Partitioning IR Program With A Scope Tree. A program has a global scope where variables are visible across the program. Other scopes should be inside existing ones. This forms directed relations between scopes: variables in the outer scope are visible to the inner scope, but not vice versa. Therefore, we build a directed scope tree to describe such relations. In the scope tree, the global scope is the root node, and other scopes are the child nodes of the scopes that they

are inside. As the semantic properties of IR tell which IRs create new scopes, we create a new node of scope when we find such an IR. We assign each node a unique id and label the IR, along with their children IRs (*i.e.*, their operands), with this id to indicate that they belong to this scope. In this way, we partition the IR program into different scopes in the tree. A variable is visible to a node if the variable is in any node long the path from the root node to the given node.

Fig. 5b shows the constructed scope tree of Fig. 5a. "Line" means the IRs translated from these lines belong to the scope or the children of the scope. Scope 1 is the global scope, which is the root node. Scope 2 and 3 are created by the structure body of `S` and function body of `main` respectively and they are child nodes of scope 1. Variables in scope 1 and 3 are visible to scope 4 as they are in the same path.

Symbol Table. We integrate the collected information of types and scopes by building symbol tables which contain the names, scopes, defined orders, and types of variables. They describe what variables (names) are available at any program location (scopes and defined orders) and how they can be used (types).

Fig. 5b shows symbol tables of each scope for Fig. 5a. Variables `s, a, c` and function `main` are defined in scope 1, the global scope, and `d` is defined in scope 2 which is the scope for structure body. There is no definition in the `IF` statement so its symbol table is empty. `TID` is the type id of the variable, which corresponds to the `TID` in the type map. `OID` is the defined order of variables and we currently use the line number as `OID` for easy demonstration. A variable is visible at a given location (*i.e.*, line number) if its scope is the ancestor node of the scope of the location and if it is defined before the location.

C. Validation

With the symbol tables, we can fix the semantic errors in the mutated test case. We call this process *validation*. We replace every invalid variable with the special string `FIXME` to indicate that this is an error to be fixed, as shown in Fig. 5a.

Specifically, we first remove any IRs that use user-defined types in the mutated part in case we cannot find the definition of these types. Then, for each `FIXME` in the mutated code, we replace it with a correct expression. We generate the expressions with the variables in the symbol tables according to their types and scopes. For example, in Fig. 5a, the original `for` statement is replaced by an `if` statement during mutation (line 6-10). The `if` statement contains a user-defined structure without definition (line 8), so we remove line 8. Finally, we replace the `FIXME`s with generated expressions.

Generating Valid Expressions. PolyGlot generates four types of expressions: a simple variable, a function call, an element indexed from an array, and a member accessed from a structure. In Fig. 5a, `"a"`, `"main()"`, `"b[1]"`, `"s.d"` are all examples of generated expressions.

First, PolyGlot infers the type of expressions containing `FIXME` and tries to figure out what type of expressions should be used for replacement. It adopts a bottom-up approach: it assigns `AnyType` to each `FIXME`, and converts `AnyType` to a more specific type when it goes up and encounters concrete operators.

TABLE I: Line of codes of different components of PolyGlot, which sum up to 7,016 lines. As we build our fuzzer on AFL, we only calculate the code that we add into AFL, which is 285 lines in the fuzzer component.

Module	Language	LOC
Frontend Generator	C++	367
	Python	1,473
Constrained Mutator	C++	1,273
Semantic Validator	C++	3,313
Fuzzer	C++	285
Others	C++/Bash	305
Total	C++/Python/Bash	**7,016**

For example, we want to fix the two `FIXME`s in the expression `"FIXME >= FIXME"` in line 10 of Fig. 5a. We assign `AnyType` to both of them. Then we go up the expression and encounter the operator `">="`, which accepts numeric types as operands, such as `int` and `short`, and outputs a result of type `bool`. As the operands are `FIXME` of `AnyType`, which can be used as any other specific type, we convert the type of `FIXME` to numeric types. Now PolyGlot needs to generate two expressions of numeric types to replace the two `FIXME`s.

Second, PolyGlot checks the symbol tables to collect all the available variables. It walks through the symbol tables of all the visible scopes in scope tree, from the global scope to the scope of the expression with `FIXME`, and collect the variables defined before the to-be-validated expression.

Third, we enumerate the possible expressions we can generate from these variables and categorize them by types. For example, from the definition `s` in line 1 in Fig. 5a, we can generate the expressions `s` and `s.d`. They are of different types so they belong to different categories.

Finally, PolyGlot randomly picks some expressions of the required type to replace `FIXME`s. If every `FIXME` of a test case can be replaced by a proper expression, the validation succeeds. The validated test case should be semantically correct and we feed it to the fuzzer for execution. If the validation fails (*e.g.*, there is no definition for a specific type), we treat the test case as semantically invalid and discard it.

One possible solution to fix `FIXME >= FIXME` at line 11 in Fig. 5a is `"b[1] >= s.d "`, where we replace the `FIXME`s with `"b[1]"` of type `short` and `"s.d"` of type `int`. `short` and `int` are of different numeric types, but `short` can be converted to `int`. Therefore, `b[1]` and `s.d` can be compared by `>=` though they are of different types.

VII. Implementation

We implement PolyGlot with 7,016 lines of code. Table I shows the breakdown.

Frontend Generation. We extend the IR format proposed in [78] by adding semantic properties. Users provide semantic annotations to help generate these properties. The frontend generator generates a parsing and a translation method from code templates written in C++ for each symbol in the BNF grammar. Then we use Bison [11] and Flex [9] to generate a parser with the parsing methods. The parser and the translation methods are compiled together to be an IR translator.

Scope Tree Construction. The scope system maintains a stack of scopes to construct the scope tree. The scope in the stack top indicates the current scope. First, it generates the global scope as the root and pushes it in the stack. Next, it walks through IRs in the IR program, labeling each IR with the id of the scope in the stack top. Meanwhile, it checks the semantic properties of the IR. If the scope system meets an IR that creates a new scope, it creates one. It sets the new scope as the child node of the scope at the stack top and pushes it to the stack. After the children of the IR are recursively processed, the scope system pops the scope out of the stack. In this way, we construct the scope tree and partition the IRs.

Builtin Variables and Functions. To improve the diversity of the generated expressions, POLYGLOT allows users to optionally add predefined builtin variables and functions of the tested programming language. These builtin variables and functions are written in the source format and added along with the initial seed corpus. POLYGLOT then analyzes these test cases and collects them as definitions. These definitions will be added to the symbol table of the global scope of every generated test case and thus used for expression generation.

Complex Expression Generation. To introduce more code structures in the test cases, we allow semantic validation to generate complex expressions. Since we have the symbol tables and the inference rule of operators, we can chain simple expressions with operators. For example, with the symbol tables in Fig. 5b, we can generate complex expressions such as `(a + b[1]) » c`, which is chaining three simple expressions (`a`, `b[1]`, `c`) with three operators (`+`, `()`, `»`). We first randomly pick an operator and then recursively generate expressions of the types of its operands. Afterward, we simply concatenate them to get a complex expression.

Fuzzer. We build POLYGLOT on top of `AFL` 2.56b. We keep the fork-server mechanism and the queue schedule algorithm of AFL and replace its test case generation module with POLYGLOT's. POLYGLOT also makes use of `AFL`'s QEMU mode, which can test binary without instrumentation. Since many programming languages are bootstrapping, which means their language processors are written in themselves, it is difficult or time-consuming to instrument these processors. Using `AFL` QEMU mode can greatly save time and effort.

User Inputs for Adoption. To apply POLYGLOT to a programming language, users need to provide: the BNF grammar, the semantic annotations and the initial corpus of test cases. The BNF grammars of most programming languages are available from either the official documents of the languages or open-source repositories [12]. The semantic annotations should describe symbols that relate to definitions, symbols that create new scopes, basic types of the languages, and the inference rules of operators. We provide a template of semantic annotations in JSON format so users can easily adjust them according to their needs. Users are free to choose the corpus that fits the tested processor. In our case, it took one of our authors 2-3 hours to collect the amentioned inputs for one language and 3-5 hours to refine them to fit in POLYGLOT.

TABLE II: 21 compliers and interpreters of 9 programming languages tested by POLYGLOT. # refers to the TIOBE index, a measurement of the popularity for programming languages [20], and - means that language is not within top 50. * in Version means the git commit hash. #Bug shows the number of reported bugs, confirmed bugs and fixed bugs from left to right.

#	Language	Target	Version	LOC(K)	#Bug
1	C	GCC	10.0	5,956	6/5/1
		Clang	11.0.0	1,578	24/3/2
4	C++	G++	10.0	5,956	4/4/2
		Clang++	11.0.0	1,578	6/0/0
7	JavaScript	V8	8.2.0	811	3/3/2
		JSCore	2.27.4	497	1/1/1
		ChakraCore	1.12.0	690	9/4/0
		Hermes	0.5.0	620	1/1/1
		mujs	9f3e141*	15	1/1/1
		njs	0.4.3	78	4/4/0
		JerryScript	2.4.0	173	5/5/4
		DukTape	2.5.0	238	1/1/1
		QuickJs	32d72d4*	89	1/1/1
8	R	R	4.0.2	851	4/4/4
		pqR	5c6058e*	845	3/1/0
9	PHP	php	8.0.0	1,269	35/27/22
10	SQL	SQLite	3.32	304	27/27/27
41	Lua	lua	5.4.0	31	12/12/12
		luajit	2.1	88	2/2/2
-	Solidity	solc	0.6.3	192	16/16/16
-	Pascal	freepascal	3.3.1	405	8/8/8
Sum	**9**	**21**			**173/136/113**

VIII. EVALUATION

Our evaluation aims to answer the following questions:

- Can POLYGLOT generally apply to different real-world programming languages and identify new bugs in their language processors? (§VIII-B)
- Can semantic validation improve POLYGLOT's fuzzing effectiveness? (§VIII-C)
- Can POLYGLOT outperform state-of-the-art fuzzers? (§VIII-D)

A. Evaluation Setup

Benchmark. To evaluate the genericity of POLYGLOT, we test 21 popular processors of nine programming languages according to their popularity [20] and variety in domains (*e.g.*, `Solidity` for smart contracts, R for statistical computation, SQL for data management). We show the target list in Table II. To understand the contributions of our semantic validation, we perform an in-depth evaluation on the representative processors of four popular languages (two statically typed and two dynamically typed): `Clang` of C, solc of `Solidity`, `ChakraCore` of `JavaScript` and php of PHP. We also use these four processors to conduct the detailed evaluation to compare POLYGLOT with five state-of-the-art fuzzers, including three generic ones (the mutation-based `AFL`, the hybrid `QSYM`, the grammar-based `Nautilus`) and two language-specific ones (`CSmith` of C and `DIE` of `JavaScript`).

Seed Corpus and BNF Grammar. We collect seed corpus from the official GitHub repository of each language processor. Additionally, we collected 71 and 2,598 builtin functions or variables for `JavaScript` and PHP respectively from [13] and [18] using a crawler script. We feed the same seeds to

```c
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3] = {1, 2, 0};
  for (a = 0; a < 39; a++)
    e = b;
  switch (c){
    while(--a){
      do{
        case 7:
          c += *(e++);
      }
      while (*e);
    }
  }
  int d = 3;
  return 0;
}
```

(a) PoC for case study 1

```c
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3] = {1, 2, 0};
  for (a = 0; a < 39; a++)
    e = b;
  if(c == 7){
    do{
      do{
          c += *(e++);
      }
      while (*e);
    }
    while(--a);
  }
  int d = 3;
  return 0;
}
```

(b) Logically equivalent program

Fig. 6: PoC and its logically equivalent program for Case Study 1. The code in line 7 to 15 should not be executed because the value of `c` is `0`. However, the development branch of Clang crashes when compiling the PoC with "-O3".

AFL, QSYM, `DIE` and POLYGLOT as initial corpus. `Nautilus` and `CSmith` do not require seed inputs. The BNF grammar POLYGLOT uses is collected and adjusted from ANTLR [12]. The official release of `Nautilus` only supports the grammars of `JavaScript` and PHP, so we further provide the grammars of C and `Solidity` to `Nautilus`.

Environment Setup. We perform our evaluation on a machine with an AMD EPYC 7352 24-Core processor (2.3GHz), 256GB RAMs, and an Ubuntu 16.04 operating system. We adopt edge coverage as the feedback and use AFL-LLVM mode [77] or AFL-QEMU mode to instrument the tested applications. We enlarge the bitmap to 1024K-bytes to mitigate path collisions [35]. Each tested target is compiled with the default configuration and with debug assertion on. We additionally patch `ChakraCore` to ignore out-of-memory errors, which can be easily triggered by valid test cases, like large-array allocations or recursive function calls, leading to lots of fake crashes and false invalid test cases in language correctness. For new bug detection, due to the limited computation resource, we tested the 21 targets for different duration, summing up to a total period of about three months. For other evaluations, we run each fuzzing instance (one fuzzer + one target) for 24h and repeat this process five times. Each fuzzing instance is run separately in a docker with 1 CPU and 10G RAM. We perform Mann-Whitney U test [67] to calculate the P-values for the experiments and provide the result in Table IV.

B. Generic Applicability and Identified Bugs

To evaluate the generic applicability of POLYGLOT, we applied it on 21 representative processors of 9 programming languages to see whether POLYGLOT can thoroughly test them and detect bugs. We only use about 450 lines of BNF grammar and 200 lines of semantic annotations on average for each of the 9 programming languages. As also mentioned in §VII, it only took one of our authors about 5-8 hours to apply POLYGLOT on each of the tested programming language.

Identified Bugs. As shown in Table II, POLYGLOT has successfully identified 173 bugs in the 21 tested processors

```c
function handler(key, value) {
    new Uint32Array(this[8] = handler);
    return 1.8457939563e-314;
}   //1.8457939563e-314 is the floating point
    //representation of 0xdeadbeef
JSON.parse("[1, 2, 3, 4]", handler);
```

Fig. 7: PoC that hijacks control flow in njs. njs crashes with `RIP` hitting `0xdeadbeef`. The bug is assigned with `CVE-2020-24349`.

```php
<?php
  $a = 'x';
  str_replace($a , array () , $GLOBALS );
?>
```

Fig. 8: PoC that triggers an invalid memory write in PHP interpreter. This kind of bug does not involve dangerous functions and can be used to escape PHP sandboxes.

of 9 programming languages, including 30 from C, 10 from C++, 26 from `JavaScript`, 35 from PHP, 16 from `Solidity`, 27 from SQL, 14 from LUA, 7 from R and 8 from Pascal. The complete and detailed information of the bugs can be found in Table VI in Appendix. All the bugs have been reported to and acknowledged by the corresponding developers. At the time of paper writing, 113 bugs have been fixed and 18 CVEs are assigned. Most of these bugs exist in the deep logic of the language processors and are only triggerable by semantically correct test cases. In the following case studies, we discuss some of the representative bugs to understand how POLYGLOT can find these bugs and what security consequences they cause.

Case Study 1: Triggering Deep Bugs in Clang. POLYGLOT identifies a bug in the loop strength reduction optimization of `Clang`. Fig. 6a shows the Proof-of-Concept (PoC), and Fig. 6b shows its logically equivalent program for understanding the semantics of the PoC easily. Fig. 11 in Appendix shows the process of how POLYGLOT turns the benign motivating example (Fig. 2a) into the bug triggering PoC. After each round of mutation, all the definitions are intact, and new code structures are introduced. Each round of validation produces a semantically correct test case. With new code structures and semantic correctness, the mutated test case keeps discovering new execution paths, which encourages POLYGLOT to keep mutating it. And we get the bug triggering PoC after 3 rounds of mutation and validation. The PoC might look uncommon to programmers, but its syntax and semantics are legitimate in C. Therefore, the PoC shows that POLYGLOT can generate high-quality inputs to trigger deep bugs in language processors.

Case Study 2: Control Flow Hijacking in njs. POLYGLOT identified many exploitable bugs, including the one shown in Fig. 7, which leads to control flow hijacking in njs. In `JavaScript`, when `JSON` parses a string, it accepts a handler to transform the parsed values. In the PoC, `JSON.parse` first parses the string `"[1, 2, 3, 4]"` into an array of four integer elements, which is denoted by `arr`. Then, the handler at line 1 runs on each of the four elements and replaces them with `1.8457939563e-314`. It also modifies `arr`, which is referred by `this` in line 2. Assigning a new element of type `function` to `arr` changes the underlying memory layout of `arr`. After the handler processes the first element, the memory layout of `arr`

TABLE III: Distribution of bugs found by evaluated fuzzers. We perform the evaluation for 24 hours and repeat it five times. We report the bugs that appear at least once. "-" means the fuzzers are not applicable to the target. PolyGlot-*st* refers to PolyGlot-*syntax*.

Target	Type SF: segmentation fault AF: assertion failure UAF: use-after-free SBOF: stack buffer overflow	PolyGlot	PolyGlot-*st*	AFL	QSYM	Nautilus	CSmith	DIE
clang	AF in parser	✗	✗	✔	✗	✗	✗	-
clang	AF in parser	✗	✗	✗	✔	✗	✗	-
clang	AF in code generation	✔	✗	✗	✗	✗	✗	-
clang	SF in optimization	✔	✗	✗	✗	✗	✗	-
ChakraCore	SF in JIT compilation	✔	✗	✗	✗	✗	-	✗
ChakraCore	AF in JIT compilation	✔	✗	✗	✗	✗	-	✗
php	UAF in string index	✔	✗	✗	✗	✗	-	-
php	SF in setlocale	✔	✔	✗	✗	✗	-	-
php	SF in zend API	✔	✔	✗	✗	✔	-	-
php	SBOF in header callback	✔	✗	✗	✗	✗	-	-
solc	SBOF in recursive struct	✔	✗	✗	✗	✗	-	-

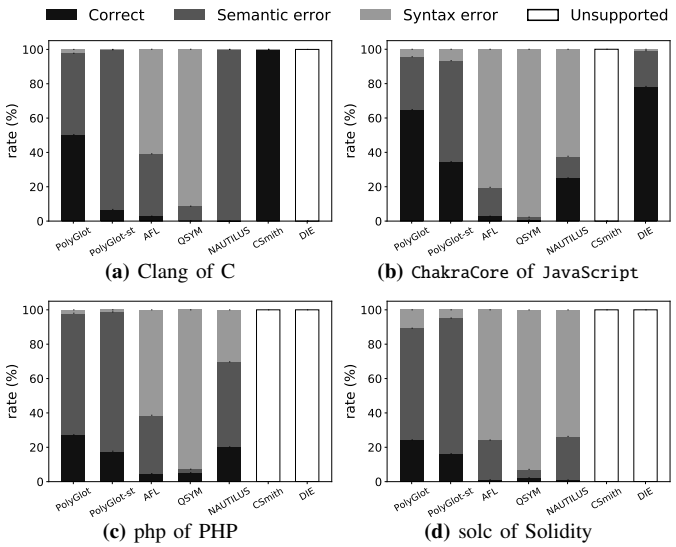


Fig. 9: Rate of language correctness of inputs generated by evaluated fuzzers for 24h. "Correct" means the inputs can be successfully executed or compiled by the language processors. "Syntax error" means the inputs contain syntactic errors. "Semantic error" means the inputs are valid syntactically but not semantically. "Unsupported" mean the fuzzer is not applicable to the target.

has changed. However, it is undetected by njs and causes a type confusion. njs still uses the old layout and mistakenly treats the first processed element, which is a user-controllable number, as a function pointer. njs then calls that function and control flow hijacking happens. If an attacker controls the `JavaScript` code, he can utilize this bug to achieve RCE.

Case Study 3: Bypassing PHP Sandbox. The PHP bugs found by PolyGlot can be used to escape PHP sandboxes. PHP sandboxes usually disable dangerous functions like `"shell_exec"` to prevent users from executing arbitrary commands. The PoCs of our PHP bugs do not involve these functions. Therefore, they are allowed to run in PHP sandboxes such as [16, 19], causing memory corruption and leading to sandbox escape [4, 5, 10].

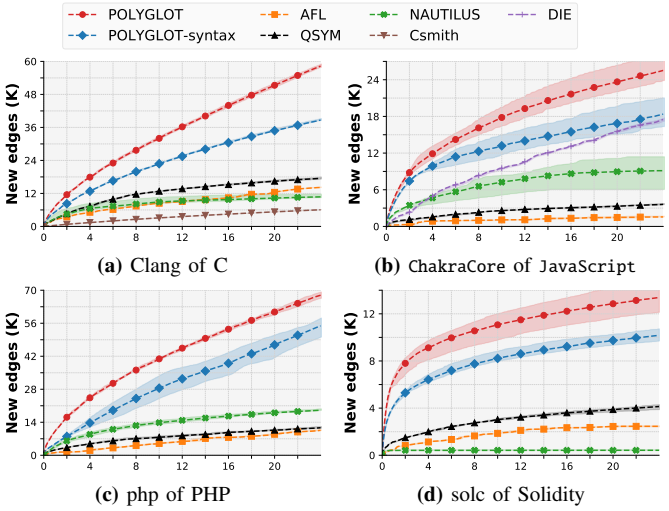
We show the PoC of one of our bugs in Fig. 8, which only uses a commonly-used and benign function `str_replace`. It triggers an out-of-boundary memory write and crashes the interpreter. With a well-crafted exploiting script, attackers can modify the benign function pointers to dangerous ones. For example, we overwrite the function pointer of `echo` to `shell_exec`. Then calling `echo("ls")`, which should simply print the string `"ls"`, becomes `shell_exec("ls")`. In this way, attackers can escape the sandbox and produce more severe damages. Actually, the security team of Google also considers bugs in PHP interpreter as highly security-related [7]. Therefore, our bugs in PHP interpreters, though not assigned with CVEs, can lead to severe security consequences.

C. Contributions of Semantic Correctness

To understand the contributions of semantic correctness in fuzzing language processors, we perform unit tests by comparing PolyGlot and PolyGlot-*syntax* which is PolyGlot without semantic validation. We compare them in three different metrics: the number of unique bugs, language validity, and edge coverage. We evaluate the number of unique bugs as it can better reflect bug finding capabilities of the fuzzers than the number of unique crashes [48]. For language correctness, we consider a test case as semantically correct as long as it can be compiled (for compilers) or executed (for interpreters) without errors. This method will treat some semantically incorrect test cases as correct ones (*e.g.*, inputs containing undefined behaviors). We plan to mitigate this problem in future work. Notice that PolyGlot-*syntax* without IR mutation is basically `AFL`, and we leave the comparison in §VIII-D.

Unique Bugs. We manually map each crash found by each fuzzer in 24 hours to its corresponding bug, and show the result in Table III. PolyGlot-*syntax* finds only two bugs in PHP, which are covered by the nine bugs PolyGlot finds in the four targets. We check the PoCs of the two PHP bugs and find the bugs are triggered by a single function call to a specific built-in function. While PolyGlot generates such function calls in correct test cases, PolyGlot-syntax generates them in incorrect ones. These function calls happen to be at the beginning of the PoCs of PolyGlot-syntax. Since the PHP interpreter parses and executes one statement per time, the bugs are triggered before the interpreter detects errors in later statements. This shows that both PolyGlot and PolyGlot-*syntax* can identify bugs triggerable by simple statements in the PHP interpreter. However, only PolyGlot detects deeper bugs in optimization in Clang and `ChakraCore` because these bugs can only be triggered by semantically correct test cases.

Language Validity. We show the details of language correctness in Fig. 9. Compared with PolyGlot-*syntax*, PolyGlot improves the language validity by 50% to 642%: 642% in `Clang`, 88% in `ChakraCore`, 54% in `php`, and 50% in `solc`. The result shows the semantic validation greatly improves the semantic correctness of the test cases. The difference in the degree of improvement results from the complexity and the accuracy of the BNF grammar of the language. In



(a) Clang of C **(b)** ChakraCore of JavaScript

(c) php of PHP **(d)** solc of Solidity

Fig. 10: Edge coverage found by evaluated fuzzers for 24h. We repeat the experiments 5 times. The solid dot lines represent the mean of the result and the shadow around lines are confidence intervals for five runs.

PolyGlot, C and JavaScript have 364 and 462 lines of BNF grammar, while PHP and Solidity have 802 and 745 respectively. Also, the BNF grammar is a superset of the real grammar the processor accepts. The mutator generates more test cases that cannot be validated due to lower accuracy in the grammar of PHP and Solidity. For example, we can use a combination of the keywords {"pure", "view", "payable"} to describe a function in Solidity as long as the same keyword does not appear twice. However, in BNF it is described as (pure|view|payable)* and then "pure pure" is legal according to the grammar. Such errors will be treated as semantic errors and cannot be fixed by PolyGlot currently.

Code Coverage. PolyGlot identifies 51%, 39%, 23%, 31% more edge paths than PolyGlot-*syntax* in Clang, ChakraCore, php, and solc respectively. We show the trend of edge coverage in 24h in Fig. 10. The increase is higher in Clang and ChakraCore than the rest two because Clang and ChakraCore perform heavy optimization (*e.g.*, ChakraCore has JIT compilation). With more semantically correct test cases, PolyGlot can reach deeper logic to explore the optimization and compilation code. As shown in Table V, PolyGlot-*syntax* executes about 2× as fast as PolyGlot, but it still achieves lower coverage. This result shows the semantically correct test cases generated by PolyGlot are more effective in exploring the deep program states.

Overall, PolyGlot outperforms PolyGlot-*syntax* in the number of unique bugs, language validity, and code coverage. As they use the same mutation strategies, they generate test cases with similar code structures. Under this condition, higher language validity further improves the fuzzing performance in various dimensions, showing the importance of semantic correctness in testing deep logic.

TABLE IV: P-values of PolyGlot *v.s.* other fuzzers. We use Mann-Whitney U test to calculate the P-values. P-values less than 0.05 mean statistical significance. The result of nearly all the experiments is statistically significant except for the language correctness compared with CSmith and DIE.

v.s. Fuzzer	Target	Coverage	Correctness	Bugs
PolyGlot-*st*	Clang	0.00596	0.00545	0.00198
	ChakraCore	0.00609	0.00583	0.00198
	php	0.00609	0.00609	0.00520
	solc	0.00609	0.00596	0.00198
AFL	Clang	0.00596	0.00545	0.00279
	ChakraCore	0.00609	0.00583	0.00325
	php	0.00609	0.00609	0.00325
	solc	0.00609	0.00596	0.00198
QSYM	Clang	0.00609	0.00485	0.00325
	ChakraCore	0.00609	0.00609	0.00325
	php	0.00609	0.00609	0.00325
	solc	0.00609	0.00596	0.00198
Nautilus	Clang	0.00609	0.00558	0.00198
	ChakraCore	0.00609	0.00609	0.00198
	php	0.00609	0.00609	0.00485
	solc	0.00609	0.00558	0.00198
CSmith	Clang	0.00609	0.998	0.00198
DIE	ChakraCore	0.00596	0.996	0.00198

D. Comparison with State-of-the-art Fuzzers

We also compare PolyGlot with five state-of-the-art fuzzers to further understand its strengths and weaknesses in testing language processors, including the mutation-based fuzzer AFL, the hybrid fuzzer QSYM, the grammar-based fuzzer Nautilus, and two language-specific fuzzers CSmith and DIE.

Unique Bugs. PolyGlot successfully identifies nine bugs in the four targets in 24 hours: two in Clang, two in ChakraCore, four in php and one in solc, as shown in Table III. AFL and QSYM only identify one bug in clang respectively. Nautilus detects one in the php interpreter, which is also covered by PolyGlot. CSmith and DIE find no bugs in 24 hours. The bugs found by AFL and QSYM exist in the parser of Clang. We check the PoCs and find them invalid in syntax: the bugs are triggered by some unprintable characters. PolyGlot does not find such bugs because its goal is to find deeper bugs with valid test cases. In fact, it does find bugs in the optimization logic of Clang such as the one in Case Study 1 (Fig. 6a), proving its effectiveness in finding deep bugs.

Language Validity. Compared with the three general-purpose fuzzers (AFL, QSYM, and Nautilus), PolyGlot improves the language validity by 34% to 10,000%, as shown in Fig. 9. Compared with the language-specific fuzzers, PolyGlot gets 53% and 83% as much as that of CSmith and DIE respectively. We investigate the result and find the reasons as follows. AFL and QSYM do not aim to improve the language validity as PolyGlot does. Nautilus uses a small number of fixed variable names and relies on name collision to generate correct input, which turns out to be less effective. CSmith and DIE perform much heavier and more specialized analyses than PolyGlot in one specific language and thus achieve higher validity in that language.

Code Coverage against General-purpose Fuzzers. As shown in Fig. 10, PolyGlot identifies 230% to 3,064% more

TABLE V: Execution speed of different fuzzers on the four tested programs within 24 hours. We repeat the experiment five times and report the average result. The number represent "Executions/Second".

Fuzzer	Clang	ChakraCore	php	solc
PolyGlot	5.39	5.56	20.55	20.08
PolyGlot-syntax	10.15	9.09	57.73	47.96
AFL	24.29	35.38	102.71	103.71
QSYM	10.39	10.60	37.94	52.95
Nautilus	40.59	66.37	115.07	185.24
CSmith	0.34	-	-	-
DIE	-	4.90	-	-

new edges than the three compared general-purpose fuzzers: up to 442% more in Clang, 542% more in php, 1,543% more in ChakraCore, and 3,064% more in solc. If we look at the execution speed of AFL, QSYM, and Nautilus (Table V), we can see that they all execute faster than PolyGlot. There are several reasons that might lead to the performance gap. First, PolyGlot puts more effort in analyzing mutated test cases and fixing semantic errors so its test case generation takes more time. Second, test cases of higher semantic correctness can be processed for a longer time, because language errors cause the execution to bail out early. As we see earlier, the test cases PolyGlot generates have a higher rate of language correctness and thus lead to slower execution. However, PolyGlot still achieves much higher coverage, showing that PolyGlot can generate high-quality test cases to effectively explore program states with a reasonable loss in efficiency of test case generation.

Code Coverage against Language-specific Fuzzers. As shown in Fig. 10, PolyGlot finds 863% more edges than CSmith in Clang and 46% more than DIE in ChakraCore. We should notice that CSmith and DIE actually have higher rate of language validity (Fig. 9). We analyze the results and find the following reasons. First, both CSmith and DIE execute more slowly than PolyGlot (Table V). This is because CSmith and DIE adopt heavier and more complex analyses than PolyGlot. Also, as discussed before, higher language validity may lead to slower execution. Second, CSmith generates test cases randomly without utilizing guidance, so it might generate similar test cases to keep exploring the same logic of the compilers. To confirm our speculation, we perform extra evaluations by comparing CSmith and PolyGlot in the same conditions: we disable the feedback guidance of PolyGlot, which is denoted by PolyGlot-$nofeedback$, as CSmith has no guidance; we randomly collect 5,000 test cases generated by PolyGlot-$nofeedback$ and CSmith to eliminate the effect of different execution speeds. We measure the language validity and code coverage in Clang and repeat the process five times. PolyGlot-$nofeedback$ gets 63.8% of language validity, while CSmith keeps its 100% correctness. 5,000 test cases of PolyGlot-$nofeedback$ and CSmith identify 672 and 1809 edges respectively. The results show our assumption that higher language validity helps explore more program states is still valid, but there are other aspects that also affect the code coverage of fuzzing (*e.g.*, feedback guidance, execution speed, code structures of test cases).

Overall, PolyGlot outperforms the three compared general-purpose fuzzers in the evaluated metrics and also outperforms the language-specific testing tools in the number of bugs and edge coverage. The fuzzing effectiveness of PolyGlot comes from both its constrained mutation and semantic validation. The mutation introduces various new code structures, while the validation further improves the quality of the test cases. Considering its generic applicability, we believe PolyGlot can save huge development efforts from developers and boost the testing of language processors.

IX. Discussion

We present several limitations of the current implementation of PolyGlot and discuss their possible solutions.

Limitation of Scope/Type System. PolyGlot relies on static analysis to collect type and scope information. Therefore, the scope system of PolyGlot only supports lexical scopes but not dymanic scope [69], and the information collected by type system on dynamically-typed programming languages might be imprecise as those types can only determined at run time. To overcome this problem, we can utilize dymanic execution to collect runtime information of the seed inputs before fuzzing to assist the analysis of PolyGlot. Also, as PolyGlot tries to be general, its scope and type system currently focus on common features shared by popular languages. To support some language-specific features, such as the ownership in Rust [68], we need to specialize PolyGlot case by case. This is not our goal, but we believe with our general IR, which carries semantic information, users can easily extend the type and scope system according to their needs.

Inconsistent Grammar. PolyGlot accepts a BNF grammar as input and generates test cases that follow the grammar. However, it still generates syntactically incorrect test cases as shown in Fig. 9, because a BNF grammar is usually a superset of the real grammar that language processors accept. For example, in C we can use `"(int|long|void)+ identifier"` to describe the grammar of variable definitions, where "+" means one or more. The "+" is intended for types like `long int` and `short int`, but invalid types like `long void` and `void int` are also valid according to the BNF grammar, which introduces incorrect test cases. To address this problem, we plan to adopt techniques that infer accurate grammar in runtime [40, 41]. Alternatively, we can try machine learning techniques to infer the accurate input grammar from test cases [37, 50, 72].

Supporting More Semantics. Currently PolyGlot improves language validity by ensuring that we use definitions of correct scopes and types. We can support more semantics shared by common programming languages to further improve semantic correctness. For example, we can use a variable only after it has been initialized, which can reduce the frequency of undefined behaviors in programming languages such as C and Pascal. Also, we can extend our symbol tables to allow variable shadowing, which allows variables of the same name to exist in different scopes. Furthermore, instead of only using the type from variable definitions, we can track when their types

change with newly assigned values. We plan to implement these features to better support different programming languages.

More Relaxed Mutation. POLYGLOT restricts its mutation to preserve language correctness. However, this restriction limits the possible definitions in the test cases POLYGLOT generates as we cannot mutate definitions. It also limits possible code structures because we hardly mutate IRs with definitions. We plan to relax the constraints in the following ways. First, we can generate and insert some definitions into the test cases. This can enrich the possible definitions of the test cases and bring more code structures. Second, we can perform test case minimization to remove definitions that are not used in the program. This also increases the possibility of mutation. For example, an IR might not be mutable because it contains a definition. If the definition is not used and removed during minimization, the IR becomes mutable.

Alternative Feedback Guidance. POLYGLOT uses code coverage as feedback guidance. The intuition is that the more code we cover, the more likely we can reach edge cases and detect bugs. However, recent research finds naive code coverage potentially harmful in finding deep bugs of language processors [53, 78]. At the early stage of fuzzing, semantically incorrect test cases can trigger new execution paths and lure the fuzzer to favor these test cases, which lowers the language validity of generated test cases. For example, we can see from §VIII that POLYGLOT achieves lower language validity (53%) than POLYGLOT without feedback (63.8%). Moreover, the frontend parsers of the processors have been well tested and are less likely to contain bugs [38, 77]. Exploring these parsers might have little help in finding new bugs. We plan to investigate this problem by trying different feedbacks, such as semantic correctness or whether the test case triggers optimization, to figure out a better guidance for testing language processors.

X. RELATED WORK

Generation-based Approaches. Generation-based fuzzing can effectively test software that require structural inputs, such as compilers and document viewers [1, 46, 51, 54, 60, 74]. They usually leverage a model or grammar, which describes the required format of the inputs, so they can efficiently generate test cases that pass the syntax check of the parsers. MoWF [54] shows how to use file format information as the model to fuzz the code beyond the parser. SQLsmith [1] generates SQL queries utilizing SQL grammar and database schemas.

However, it can be nontrivial to get the model or grammar. For example, the tested application is closed-source and has no public documents. Recent works propose methods to infer the structures of the inputs by static analysis or machine learning on an initial seed corpus [24, 39, 45, 61]. Viide *et al*. [61] proposes a model inference approach to assist fuzzing. Osbert *et al*. [24] utilizes a set of test cases and the black-box access to the tested binary to construct a context-free grammar of the language. AUTOGRAM [45] uses dynamic taints to produce readable and accurate input grammars.

Considering the infinite input space, blindly generating test cases is still inefficient for exploring program states. Therefore, researchers also propose utilizing feedback from execution to guide test case generation. Apollo [47] measures the difference in execution time to favor generated SQL queries. Nautilus [22] adopts code coverage as feedback to decide whether to keep its generated test cases for mutation.

Programming language testing further requires the semantic correctness of the inputs. Improving the semantic correctness of the generated test case greatly helps fuzzers detect deeper bugs in language processors [33, 74]. CodeAlchemist [43] proposes semantics-aware assembly to synthesize semantics-correct `JavaScript` test cases. CSmith [74] specializes its analysis for C semantics and produces completely correct test cases. Dewey *et al*. uses constraint logic programming to specify syntactic features and semantic behaviors in test case generation [33, 34]. The method relies on symbolic executions and complex constraint programming.

POLYGLOT differs from the aforementioned works in the following aspects: POLYGLOT adopts grammar for mutation instead of pure generation so it can fully utilize coverage guidance; POLYGLOT is generic and easy to apply on different language processors.

Mutation-based Approaches. Mutation-based fuzzing is effective in exploring deep logic of tested programs [38, 76, 77]. Unlike generation-based ones, mutation-based fuzzers usually require an initial corpus to run. They perform random mutation on existing test cases to generate new ones. If a test case triggers a new execution path, it will be considered as useful and saved for further mutation. In this way, fuzzers quickly reach the deep logic and explore more program states. AFL [77] adopts coverage feedback as guidance and performs random bitflip mutation. As naive bitflip mutation can hardly pass complicated checks such as magic numbers, existing fuzzers [23, 25, 29, 30, 36, 56, 58, 76] adopt symbolic execution or taint analysis to overcome the problem. Driller [58] performs selective concolic execution while QSYM [76] integrates the symbolic emulation with the native execution.

To fully utilize computation power, researchers try to find a better feedback guidance other than naive code coverage [27, 28, 49, 65]. AFLGo [27] introduces directed greybox fuzzing with the objective of reaching a given set of target program locations efficiently. TaintScope [64] uses checksum as feedback guidance to help fuzz file segments. SAVIOR [31] prioritizes its concolic execution towards the locations with potential vulnerabilities. Ijon [21] annotates the data that represent the internal program states to guide the fuzzer.

However, the aforementioned mutation-based fuzzers are unaware of the input structures. Their effectiveness greatly reduces when highly structural inputs are required. Recent fuzzers try to learn the structures [26, 32, 62, 75]. DIFUZE [32] leverages static analysis to identify the input structures of kernel drivers. GRIMORE [26] performs large-scale mutation using grammar-like combinations to synthesize structured inputs without users' help. SLF [75] infers the relation between input validity checks and input fields to generate valid seed inputs.

Alternatively, researchers propose advanced mutation on a higher level than bits and bytes [42, 44, 52, 55, 63, 71, 73, 78]. LangFuzz [44] and Superion [63] accept a grammar to translate test cases to AST and then mutate the AST. Fuzzilli [42] and Squirrel [78] design their own IRs for mutation and semantic analysis in `JavaScript` and SQL respectively.

Compared with these fuzzers, POLYGLOT adopts light-weight analyses to efficiently generate valid inputs that pass the syntactic and semantic checks of language processors. Meanwhile, it keeps its generic applicability by basing its analysis on a uniform IR.

XI. CONCLUSION

We present POLYGLOT, a generic fuzzing framework that generates high-quality inputs for testing processors of different programming languages. We applied POLYGLOT on 21 processors of 9 languages and successfully identified 173 new bugs. Our evaluation shows POLYGLOT is more effective in testing language processors than existing fuzzers with up to 30× improvement in code coverage.

References

[1] SQLSmith. https://github.com/anse1/sqlsmith, 2016.

[2] Domato, A DOM fuzzer. https://github.com/googleprojectzero/domato, 2017.

[3] List of programming languages. https://en.wikipedia.org/wiki/List_of_programming_languages, 2017.

[4] Dismissed PHP flaw shown to pose code execution risk. https://portswigger.net/daily-swig/dismissed-php-flaw-shown-to-pose-code-execution-risk, 2019.

[5] PHP 7.0-7.4 disable_functions bypass. https://github.com/mm0r1/exploits/tree/master/php7-backtrace-bypass, 2019.

[6] THE STORY OF TWO WINNING PWN2OWN JIT VULNERABILITIES IN MOZILLA FIREFOX. https://www.thezdi.com/blog/2019/4/18/the-story-of-two-winning-pwn2own-jit-vulnerabilities-in-mozilla-firefox, 2019.

[7] Bugs in PHP interpreter found by OSS-FUZZ. https://bugs.chromium.org/p/oss-fuzz/issues/list?q=-status%3AWontFix%2CDuplicate%20-component%3AInfra%20PHP&can=1, 2020.

[8] Data type. https://en.wikipedia.org/wiki/Data_type, 2020.

[9] Flex, the fast lexical analyzer generator. https://github.com/westes/flex/, 2020.

[10] From Web to Pwn - FFI Arbitrary read/write without FFI::cdef or FFI::load. http://blog.huntergregal.com/2020/07/from-web-to-pwn-ffi-arbitrary-readwrite.html, 2020.

[11] Gnu bison. https://www.gnu.org/software/bison/, 2020.

[12] Grammars written for ANTLR v4. https://github.com/antlr/grammars-v4, 2020.

[13] JavaScript: Standard built-in objects. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects, 2020.

[14] Memory unsafety problem in safe Rust. https://github.com/rust-lang/rust/issues/69225, 2020.

[15] Miscompilation: for range loop reading past slice end. https://github.com/golang/go/issues/40367, 2020.

[16] Online PHP Sandbox. https://wtools.io/php-sandbox/, 2020.

[17] OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz, 2020.

[18] PHP: Internal (built-in) functions. https://www.php.net/manual/en/functions.internal.php, 2020.

[19] PHP Sandbox, Test your PHP code with this code tester. https://sandbox.onlinephpfunctions.com/, 2020.

[20] Tiobe index for august 2020. https://www.tiobe.com/tiobe-index/, 2020.

[21] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612, 2020.

[22] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[23] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[24] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 95–110, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 818–825. IEEE, 2012.

[26] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: Synthesizing structure while fuzzing. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1985–2002, USA, 2019. USENIX Association.

[27] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[28] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[29] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[30] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[31] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. Savior: Towards bug-driven hybrid testing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[32] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[33] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 725–730, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using clp (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 482–493. IEEE, 2015.

[35] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[36] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

[37] P. Godefroid, H. Peleg, and R. Singh. Learn fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59, 2017.

[38] Google. Honggfuzz, 2016. https://google.github.io/honggfuzz/.

[39] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289*, 2018.

[40] Rahul Gopinath, Björn Mathis, Matthias Höschele, Alexander Kampmann, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *ArXiv*, abs/1810.08289, 2018.

[41] Rahul Gopinath, Björn Mathis, and Andreas Zeller. Inferring input grammars from dynamic control flow. *arXiv preprint arXiv:1912.05937*, 2019.

[42] Samuel Groß. Fuzzil: Coverage guided fuzzing for javascript engines. *Master thesis, TU Braunschweig*, 2018.

[43] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. 2019.

[44] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 38, USA, 2012. USENIX Association.

[45] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725. IEEE, 2016.

[46] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 259–269, New York, NY, USA, 2018. Association for Computing Machinery.

[47] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. APOLLO: Automatic Detection and Diagnosis of Performance Regres-

sions in Database Systems (to appear). In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[48] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[49] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[50] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1044–1051, 2019.

[51] MozillaSecurity. funfuzz. https://github.com/MozillaSecurity/funfuzz, 2020.

[52] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340, 2019.

[53] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[54] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016.

[55] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[56] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[57] Chris Rohlf and Yan Ivnitskiy. Attacking clientside jit compilers. *Black Hat USA*, 2011.

[58] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[59] Ken Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8), August 1984.

[60] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[61] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. Experiences with model inference assisted fuzzing. In *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies*, WOOT'08, USA, 2008. USENIX Association.

[62] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[63] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[64] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512, 2010.

[65] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. NDSS, 2020.

[66] Wikipedia contributors. Backus-naur form — Wikipedia, the free encyclopedia, 2020. [Online; accessed 1-September-2020].

[67] Wikipedia contributors. Mann-whitney u test — Wikipedia, the free encyclopedia, 2020. [Online; accessed 31-August-2020].

[68] Wikipedia contributors. Rust (programming language) — Wikipedia, the free encyclopedia, 2020. [Online; accessed 23-December-2020].

[69] Wikipedia contributors. Scope (computer science) — Wikipedia, the free encyclopedia, 2020. [Online; accessed 23-December-2020].

[70] Wikipedia contributors. Translator (computing) — Wikipedia, the free encyclopedia, 2020. [Online; accessed 1-September-2020].

[71] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware mutations for testing smt solvers. *arXiv preprint arXiv:2004.08799*, 2020.

[72] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. Reinam: reinforcement learning for input-grammar inference. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 488–498, 2019.

[73] Dingning Yang, Yuqing Zhang, and Qixu Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1070–1076. IEEE, 2012.

[74] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, New York, NY, USA, 2011.

[75] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang. Slf: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 712–723, 2019.

[76] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, USA, 2018.

[77] Michal Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl, 2019.

[78] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, USA, November 2020.

TABLE VI: Detected bugs. POLYGLOT found 173 bugs, including 6 from GCC, 24 from Clang, 4 from G++, 6 from Clang++, 3 from v8, 1 from JSCore, 9 from ChakraCore, 1 from Hermes, 1 from mujs, 4 from njs, 5 from JerryScript, 1 from DukTape, 1 from QuickJs, 4 from R, 3 from pqR, 35 from php, 27 from SQLite, 12 from Lua, 2 from LuaJit, 16 from solc, 8 from freepascal.

Bug Type: **UAF**: use-after-free. **BOF**:buffer overflow of Global (**G**), Heap (**H**), and Stack (**S**).
SEGV: potential exploitable vulnerability. **AF**: assertion failure. **LEAK**: memory leak.

Bug Status: ⋆: Fixed, ●: Verified, ◇: Open, †: CVE

ID	Type	Function	Reference
SQLite v3.31, 300K LoC			
1	UAF	resetAccumulator	† 2020-13871
2	SBOF	sqlite3_str_vappendf	† 2020-13434
3	SEGV	sqlite3ExprCodeTarget	† 2020-13435
4	HBOF	multiSelectOrderBy	† 2020-15358
SQLite v3.32 (development), 304K LoC			
5	UAF	sqlite3BlobCompare	⋆ da5a09b
6	UAF	resetAccumulator	⋆ 7c6d876
7	SEGV	sqlite3VdbeExec	⋆ 0899cfu
8	SEGV	sqlite3VdbeExec	⋆ 1f6f353
9	SEGV	sqlite3VdbeCursorMoveto	⋆ e5504e9
10	SEGV	moveToRoot	⋆ f7d8908
11	SEGV	sqlite3Select	⋆ b706351
12	SEGV	sqlite3VdbeCursorMoveto	⋆ 5829597
13	SEGV	isAuxiliaryVtabOperator	⋆ 4374860
14	AF	flattenSubquery	⋆ e367f31
15	AF	sqlite3VdbeExec	⋆ 9fb26d3
16	AF	sqlite3ExprCodeIN	⋆ fd1bda0
17	AF	selectExprDefer	⋆ 7aa91ab
18	AF	Where clause	⋆ e12a0ae
19	AF	sqlite3VdbeExec	⋆ c4c5648
20	AF	Vdbe Memory	⋆ d165ad7
21	AF	sqlite3WhereBegin	⋆ 82b588d
22	AF	clearSelect	⋆ 618156e
23	AF	OrderBy Optimize	⋆ 41c1456
24	AF	xferOptimization	⋆ f07d71b
25	AF	Btree Cursor	⋆ aa43786
26	AF	sqlite3IsLikeFunction	⋆ b985f0b
27	AF	sqlite3VdbeExec	⋆ 9d36667
GCC v10.0.1, 5956K LoC			
28	SEGV	gimplify_compound_expr	⋆ 93576
29	AF	Nonlocal Reference	● 93572
30	AF	force_constant_size	● 93573
31	AF	c_expr_sizeof_expr	◇ 93574
32	AF	extended_tree	● 93577
33	AF	gimple_call_arg	● 93631
G++ v10.0.1, 5956K LoC			
34	SEGV	dump_parameters	● 93788
35	AF	pop_local_binding	⋆ 93752
36	AF	Output constructor	● 93753
37	AF	cc1plus	⋆ 93789
Clang v11.0.0, 1578K LoC			
38	SEGV	getExprLoc	◇ 44729
39	SEGV	isa_impl_wrap	◇ 44735
40	SEGV	ActOnFinishFullExpr	◇ 44741
41	SEGV	usePhysReg	● 44750
42	SEGV	VisitBuiltinCallExpr	◇ 44756
43	AF	CheckListElementTypes	◇ 44738
44	AF	ParseSwitchStatement	◇ 44740
45	AF	ActOnForStmt	◇ 44732
46	AF	VisitCastExpr	◇ 44742
47	AF	DoMarkVarDeclReferenced	◇ 44744
48	AF	eliminateRegSequence	⋆ 44749
49	AF	getASTRecordLayout	◇ 44734
50	AF	cast_retty	◇ 44755
51	AF	ImplicitMember	◇ 44737
52	AF	VisitBuiltinCallExpr	◇ 44757
53	AF	isIncompleteDeclExternC	◇ 44758
54	AF	udivrem	◇ 44831
55	AF	removeDecl	◇ 44832
56	AF	getIndirectResult	◇ 44850
57	AF	getCommonPtr	◇ 44867
58	AF	TailRecursionEliminator	⋆ 46125
59	AF	getTypeSizeInBits	◇ 46211
60	AF	getAccessTagInfo	◇ 46388
61	SEGV	Inifite loop	◇ 46262
Clang++ v11.0.0, 1578K LoC			
62	AF	ParenExpression	◇ 44924
63	AF	HasAccess	◇ 44925
64	AF	LookupTemplateName	◇ 44926
65	AF	ImplicitConversion	◇ 44927
66	AF	CreateOverloadedBinOp	◇ 44928
67	AF	PushDeclContext	◇ 44940

ID	Type	Function	Reference
V8 v8.2.0, 811K LoC			
68	SEGV	TimeFormatConstructor	⋆ 1052019
69	AF	DeclarationScope	⋆ 1060023
70	SEGV	monkey-patched	◇ 1098588
JaveScriptCore v2.27.4, 497K LoC			
71	SEGV	finishCreation	⋆ 208040
ChakraCore v1.12.0, 690K LoC			
72	SEGV	ReparseAsmJsModule	● 6472
73	SEGV	FlowGraph::Build	● 6473
74	AF	MapStFldHelper	● 6474
75	SEGV	CloneBlockData	◇ 6482
76	AF	FlowGraph Destruct	◇ 6483
77	AF	GetArgCount	◇ 6485
78	AF	CallFunction	◇ 6486
79	SEGV	Optimize	● 6487
80	SEGV	EmitTopLevelStatement	◇ 6488
Hermes v0.5.0, 620K LoC			
81	AF	operationWalk	⋆ 279
MuJs 9f3e141, 15K LoC			
82	UAF	jsR_run	⋆ 136
njs v0.4.3, 78K LoC			
83	SEGV	JSON Stringify	† 2020-24348
84	SEGV	njs_lvlhsh_bucket_find	† 2020-24347
85	SEGV	njs_value_property	† 2020-24349
86	UAF	JSON Parse	† 2020-24346
JerryScript v2.4.0, 173K LoC			
87	UAF	jerry-core	† 2020-24344
88	AF	Typedarray	⋆ 3975
89	AF	JSON parse	⋆ 3950
90	AF	JSON parse	⋆ 3945
91	AF	Statement parser	⋆ 3944
DukTape v2.5.0, 238K LoC			
92	AF	duk_hobject_getprop	⋆ 2338
QuickJs 32d72d4, 89K LoC			
93	LEAK	JS_FreeRuntime	⋆ c389f9594
solc v0.6.3, 193K LoC			
94	SBOF	ReferencesResolver	⋆ 8266
95	AF	calldataEncodedSize	⋆ 8275
96	AF	encodeToMemory	⋆ 8276
97	AF	Expression Assignment	⋆ 8277
98	AF	storeValue	⋆ 8278
99	AF	isDynamicallyEncoded	⋆ 8279
100	AF	interfaceType	⋆ 8280
101	AF	interfaceType	⋆ 8282
102	AF	TypeChecker::visit	⋆ 8283
103	AF	memoryDataSize	⋆ 8286
104	AF	ConstStateVar	⋆ 8296
105	AF	copyToStackTop	⋆ 9272
106	SEGV	SourceLocation	⋆ 9404
107	SEGV	RationalNumber	⋆ 9434
108	SEGV	SourceLocation	⋆ 9441
109	SBOF	interfaceType	⋆ 9443
Lua v5.4.0, 31K LoC			
110	UAF	luaD_call	† 2020-15888
111	HBOF	getobjname	† 2020-15889
112	SEGV	changedline	† 2020-15945
113	SBOF	luaO_pushvfstring	† 2020-24342
114	UAF	luaD_call	† 2020-24371
115	SEGV	lua_getlocal	† 2020-24370
116	SEGV	lua_traceexec	† 2020-24369
117	SBOF	luaO_pushvfstring	⋆ eb41999
118	HBOF	luaT_adjustvarargs	⋆ 6298903
119	HBOF	luaD_pretailcall	⋆ eb41999
120	HBOF	luaH_get	⋆ 6298903
121	UAF	lua_checkstack	⋆ 34affe7
LuaJit v2.1, 88K LoC			
122	SEGV	lj_err_run	† 2020-15890
123	SEGV	lj_err_run	† 2020-24372
R v4.0.2, 851K LoC			
124	HBOF	printing system	⋆ 17867
125	HBOF	printAttributes	⋆ 17870
126	SEGV	writelines	⋆ 17876
127	SEGV	model.matrix	⋆ 17879

ID	Type	Function	Reference
pqR 5c6058e, 845K LoC			
128	SEGV	Primitive	● issue-43
129	SEGV	do_dotcall_e	◇ issue-44
130	SEGV	plot	◇ issue-45
PHP v8.0.0, 1269K LoC			
131	SBOF	header callback	◇ 79207
132	UAF	zend_hash_packed_to_hash	● 79209
133	SEGV	php_var_export_ex	● 79214
134	SEGV	zend_hash_next_index_insert	⋆ 79258
135	SBOF	php_array_element_dump	⋆ 79259
136	AF	zend_dispatch_try	⋆ 79777
137	AF	zend_closure_get_debug_info	⋆ 79778
138	AF	zend_wrong_string_offset	⋆ 79779
139	HBOF	zend_array_destroy	◇ 79781
140	SEGV	php_str_replace_common	⋆ 79783
141	UAF	slow_index_convert	⋆ 79784
142	AF	zend_array_destroy	● 79788
143	AF	zend_ast_evaluate	⋆ 79790
144	AF	_get_zval_ptr_cv_BP_VAR_RW	⋆ 79791
145	AF	zend_array_destroy	⋆ 79792
146	UAF	fetch imension_address	⋆ 79793
147	AF	zend_gc_delref	● 79813
148	SEGV	zend_string_release_ex	⋆ 79815
149	AF	_zend_is_inconsistent	● 79816
150	SBOF	zval_get_tmp_string	⋆ 79817
151	UAF	ZEND_VM_HANDLER marco	⋆ 79818
152	SEGV	zend_std_write_property	◇ 79819
153	SEGV	zend_get_properties_for	⋆ 79821
154	SEGV	php_pcre_replace_array	⋆ 79829
155	LEAK	php_var_dump	⋆ 79830
156	AF	increment_function	● 79831
157	SEGV	property_info_for_slot	⋆ 79832
158	SEGV	php_str_replace_in_subject	◇ 79835
159	SEGV	concat_function	◇ 79836
160	SEGV	zend_get_type_by_const	◇ 79837
161	SEGV	zend_mm_alloc_small	◇ 79838
162	AF	array_walk	⋆ 79839
163	AF	zend_get_property_offset	⋆ 79862
164	LEAK	zend_hash	⋆ 79947
165	LEAK	zend_string	⋆ 79951
freepascal v3.3.1, 405K LoC			
166	AF	tmoddivnode.simplify	⋆ 37449
167	AF	TSparcReader.BuildOperand	◇ 37459
168	AF	get_funcretloc	⋆ 37460
169	AF	tinlinenode.handle_str	⋆ 37462
170	AF	replaceloadnodes	⋆ 37475
171	AF	GetCopyAndTypeCheck	◇ 37476
172	AF	searchcsedomain	⋆ 37477
173	AF	GetCopyAndTypeCheck	⋆ 37508

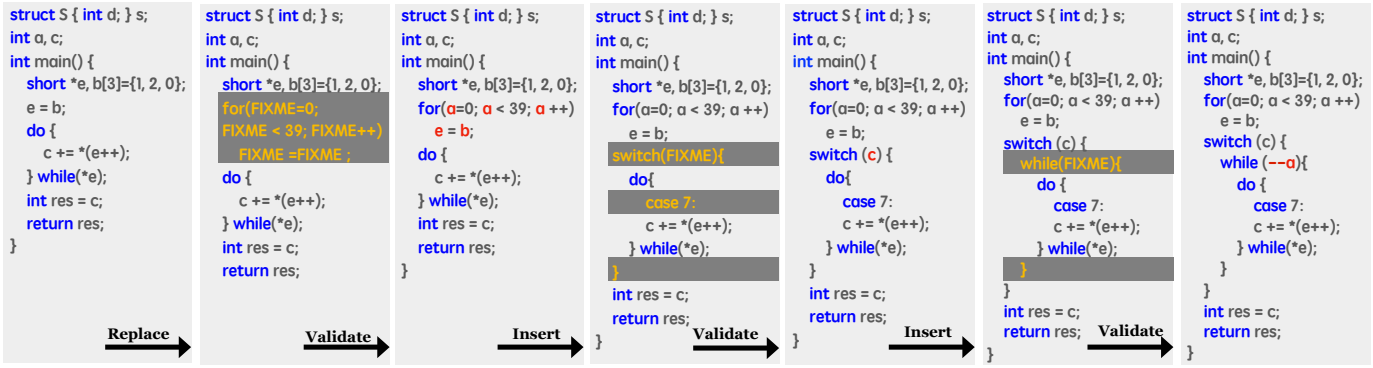


Fig. 11: The process of how POLYGLOT generates the PoC for Case Study 1. We perform three rounds of mutation and validation on the motivating example of Fig. 2a. Each round of mutation introduces new code structures (*e.g.*, the first replacement introduces a `for` statement), and each round of validation generates correct expressions to fix the semantic errors.